UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Diseño y desarrollo de arquitectura de aplicación multinivel para Android

Máster en Ingeniería Informática

Autor: Cortijo Navascués, Hugo
Tutor: Martinez Muñoz, Gonzalo

septiembre de 2015

RESUMEN

Uno de los grandes problemas a la hora de desarrollar un software es la inevitable filtración de lógica de negocio en la interfaz de usuario. Esto provoca que el software no pueda ser probado con un conjunto de tests automatizados porque cambios en la lógica provocan la mayoría de veces cambios en la interfaz. La utilización de buenas prácticas y arquitecturas a la hora de diseñar una solución software fomentan la separación de funcionalidades. Se ha realizado un estudio previo de las arquitecturas, hexagonal y de cebolla, cuyo objetivo es no depender de la interfaz de usuario, de forma que la lógica de negocio sea comprobable a través de test independientes de la interfaz de usuario o fuentes externas. Tras analizar los aspectos más importantes de estas arquitecturas se han aplicado en el proceso de digitalización de la empresa IluminaciónFM.

En este sentido en el presente trabajo se ha desarrollado una aplicación con una arquitectura multinivel y comprobable en todas sus capas que da respuesta a las necesidades de la empresa Iluminación FM. Iluminación FM es una empresa de servicios de iluminación cinematográfica y maquinista, con la que se podrá llevar a cabo cualquier producción audiovisual. Tras 50 años trabajando en este sector, estos últimos 10 años han ido digitalizando de forma parcial sus procedimientos de estimación de material y personal para llevar a cabo grandes proyectos cinematográficos. El procedimiento que llevan a cabo una vez que hablan con el cliente, fijan la localización del rodaje y tiempo de permiso de rodajes es el siguiente:

- Primero se dirige uno de los empleados a la localización del rodaje y tiene que rellenar un documento denominado ***Parte de Localización Técnica*** a papel, incluido en el **Anexo I**. En el se detallan aspectos relevantes para un presupuesto inicial (personal y material necesario).
- Una vez que el empleado termina de hacer las anotaciones a papel regresa a la oficina donde entrega el ***Parte de Localización Técnica*** y se digitaliza mediante la herramienta ***Micosoft Excel***.

Este procedimiento lleva a una pérdida considerable de tiempo y además la mayoría de los empleados no terminan de completar las anotaciones en papel y como consecuencia una mala estimación del presupuesto inicial. Esto provoca que en algunos casos haya una pérdida de proyectos.

Para evitar esta situación en **Iluminación FM** se ha desarrollado una aplicación móvil para Android y una API REST en Ruby on Rails que de soporte a la aplicación.

En este documento se plasma como surge esta necesidad de digitalizar uno de los procedimientos de la empresa **Iluminación FM**. Una vez definido estos aspectos previos se concretan las necesidades del usuario para desarrollas una solución. Dicha solución consta de un análisis previo de diseño de una arquitectura cuyo objetivo principal es la elaboración de una aplicación móvil Android a nivel de codificación flexible, robusta, eficaz y duradera. Una vez expuesto los niveles que conforman la arquitectura de la aplicación móvil, se detallarán otros aspectos técnicos que resuelven los requisitos como son la arquitectura de todo el sistema, su seguridad, diseño de interfaz gráfica, API Rest y la base de datos. Además para mejorar el tiempo de computación a la hora de generar los Partes de Localización Técnica en formato pdf se

ha optado por la utilización de colas reduciendo la carga de proceso de peticiones por parte del servidor.

Tras definir la aplicación se explicará la solución tomada y que técnicas y/o librerías se han utilizado y por qué, dando una explicación detallada de los motivos.

INDICE

GLOSARIO

Notificaciones Push	**Servicio proporcionado por Google Cloud Messaging para controlar el envío de notificaciones a un dispositivo móvil.**
Backend	Servidores que proporcionan los servicios.
URL	Unifor Resource Locator. Dirección única de un recurso en Internet.
Smartphones	Teléfono móvil capaz de realizar actividades semejantes a una computadora.
Tablet	Dispositivo con mayores dimensiones que un Smartphone.
XML	Extensible Markup Languaje. Lenguaje de etiquetado para el intercambio de datos.
JSON	Javascript Object Notation. Formato ligero de intercambio de datos.
Ruby On Rails	Framework escrito en ruby siguiendo la arquitectura MVC.
MVP	Patrón de diseño software que separa los datos y lógica de la interfaz del usuario.
API REST	Es un tipo de arquitectura de desarrollo web para crear APIs para servicios orientados a Internet.
Colas de mensaje	Intermediario entre clientes y servidores.
RabbitMQ	Colas de mensaje
NoSQL	Base de datos no relacionales.
MongoDB	Base de datos no relacional que permite indexar campos para mejorar el tiempo de consulta.
GitHub	Sistema de control de versiones.
Framework	Módulos de software concretos que sirvern para la organización y desarrollo de nuevo software.
Regla de Dependencia	Dependencias de código representadas en capas, sólo capas exteriores pueden tener dependencias de capas inferiores.
Material Design	Presentado en el Google I/O 2014 es un concepto, una filosofía y unas pautas enfocadas en el diseño utilizado en Android, pero también se pueden incorporar a la web y a cualquier otra plataforma.
Start up	Termino utilizado actualmente para referirse a empresas que emprenden un nuevo negocio.
MitM	En criptografía es un ataque que permite leer y modificar los mensajes que se envían entre entidades.
Listener	A nivel de código, objeto que recibe y maneja un evento.
Callbacks	A nivel de código, si se produce un caso llamar a una función pasada como parámetro.

INTRODUCCIÓN

1.1 Motivación y definición del problema

La empresa Iluminación FM, dedicada al alquiler de material cinematográfico tiene la necesidad de automatizar y digitalizar sus procesos y documentos generados a la hora de dar un presupuesto. El procedimiento que siguen hasta la fecha consiste en visitar la localización donde se realizará el rodaje y hacer una estimación del material necesario para el rodaje. Este proceso se lleva a cabo mediante anotaciones en papel que posteriormente se llevan a la central donde se transcriben a una hoja de cálculo. Lo que supone una pérdida considerable de tiempo. Además este proceso puede llevar en algunos casos a tener pérdidas de información lo que dificulta adicionalmente la correcta realización del presupuesto.

En este TFM se propone como solución la realización de una aplicación móvil para Android que permita recabar la información importante del proyecto a realizar de manera sencilla. De forma que se transfiera automáticamente al sistema y que permita dar un presupuesto inicial.

Uno de los grandes problemas a la hora de desarrollar un software es la inevitable filtración de lógica de negocio en la interfaz de usuario. Esto provoca que el software no pueda ser probado con un conjunto de tests automatizados porque cambios en la lógica provocan la mayoría de veces cambios en la interfaz. La primera solución que se suele utilizar es la de incluir una capa más en la arquitectura con el objetivo de separar la capa de lógica de negocio de la capa de interfaz. Sin embargo, es complejo determinar mecanismos que detecten que se está incorporando lógica de negocio en la interfaz gráfica del usuario.

Por otro lado, cualquier persona con nociones de programación es capaz de crear una aplicación que satisfaga las necesidades del usuario. Sin embargo, es necesario algo más que satisfacer las necesidades de usuario para realizar un producto de calidad. Para que un producto software sea de calidad es deseable que sea independiente de frameworks, de interfaz de usuario, de conexiones a base de datos o de cualquier otro agente externo. Si se consiguen estos objetivos estamos se conseguirá que el software sea comprobable, es decir, que se pueda validar y verificar mediante una batería de tests de la lógica de negocio y de todos los módulos externos a ella.

1.2 Objetivos del proyecto

El desarrollo de un software de calidad es una terea difícil y compleja. No consiste sólo, como hemos dicho, en satisfacer los requisitos del usuario. También debe ser flexible a cambios y actualizaciones, así como comprobable, robusto y fácil de mantener. Estos objetivos serán detallados a lo largo de este documento. Por lo tanto el objetivo principal de este proyecto es la realización de un diseño y desarrollo de una arquitectura para dispositivos móvil independiente de sistema operativo (Android, iOS, etc) que dé solución a las necesidades de la empresa Iluminación FM.

Además permitirá poder comercializar esta aplicación para otras empresas que necesiten recabar información relacionada con su negocio.
Los objetivos concretos son:

- Diseño de la arquitectura software para la aplicación móvil identificando tres capas o niveles principales. Nivel de Aplicación donde se incluye el patrón de diseño Modelo-Vista-Presentador (***MVP***). Nivel de Dominio donde ocurre la lógica de negocio. Y para finalizar la Capa de Datos que provee de información a la aplicación.
- API Rest y base de datos no relacional **MongoDB** que dará soporte a la aplicación móvil.

Por lo tanto como primera versión se ha diseñado una arquitectura denominada con en el término del desarrollo software "limpia" y acuñado por Martin Fowler (Uncle Bob).

2. ESTUDIO DEL ESTADO DEL ARTE

Para la elaboración de este proyecto se han tenido en cuenta las arquitecturas más de moda en los últimos años.

2.1 Arquitectura Hexagonal

El término fue acuñado por *Alistair Cockburn* [1]. También denominada *Puertos y Adaptadores*, su intención es poder aislar la capa de dominio de elementos externos como interfaz de usuario, base de datos, servicios de red. La arquitectura hexagonal no es una nueva forma de cómo programar con dependencias a un framework, sino es una forma de describir buenas prácticas que hagan que nuestro proyecto no tenga deuda técnica. Por deuda técnica entendemos como la deuda que se paga por malas decisiones tomadas previamente y, que si se comenten al principio del proyecto, se transforman en problemas que pueden acarrear un gran coste en fases posteriores del proyecto. Respecto a que se llame hexagonal es arbitrario ya que dependiendo de las necesidades del proyecto podrá tenar más o menos lados. La idea consiste en que la aplicación funciona como un núcleo, independiente de la interfaz de usuario, de puntos de

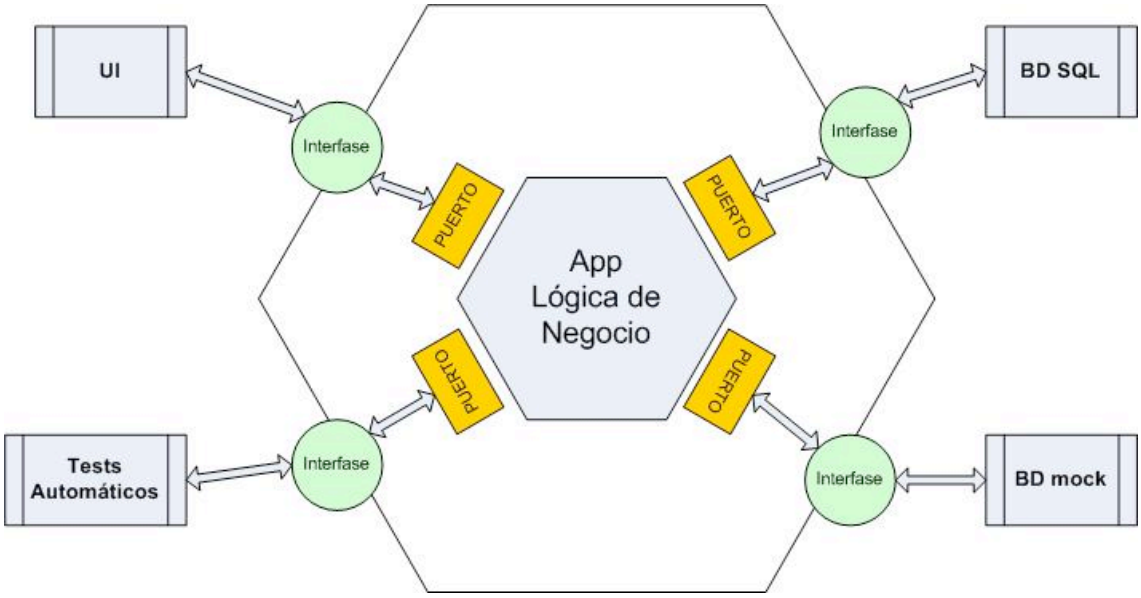

Figura 1. Arquitectura hexagonal.

persistencia y de otros servicios. Para acceder a la lógica de negocio esta tiene definidos puertos e interfaces que pueden ser adaptados (implementar las interfaces) por los diferentes bloques y así no tener conocimiento del origen de la conexión (véase Figura 1).

Como idea general, las capas se comunican unas con otras a través de puertos e implementaciones de estas (adaptadores). Cada capa tiene su propio límite y en ese límite encontramos los puertos o interfaces. Para que una capa se pueda comunicar con el núcleo o viceversa se deben de implementar dichas interfaces. Como podemos comprobar, gracias a las interfaces, desacoplamos el código entre capas por lo que crear interfaces en puntos clave de nuestra aplicación permite crear múltiples adaptaciones.

La arquitectura hexagonal no sólo trata de comunicación entre capas dentro de la propia aplicación (interfaces, implementación de interfaces) sino también tiene una frontera entre la aplicación y el framework, y entre la aplicación y el mundo exterior (otras

aplicaciones, servicos, etc) como podemos ver en la Figura 1 (hexágono externo). Por lo que también hay que definir como se comunica nuestra lógica de negocio (hexágono interno) con el mundo exterior y eso se hace a través de los casos de uso o clases que determinan las acciones que se pueden realizar. Los casos de uso son útiles a la hora de clarificar ideas entre los equipos de desarrollo, permitiéndoles planificar en el tiempo nuevos casos de uso. Para definir los casos de uso tenemos tres actores que llevan a buen puerto el procesamiento del caso de uso:

- **Caso de uso**: Es una definición explícita de cómo puede ser utilizada la aplicación.
- **Transmisor**: Es quien acepta un caso de uso, realiza alguna lógica e instancia un *Gestor* para ese caso de uso.
- **Gestor**: Es el Gestor quien ejecuta la lógica de negocio para cumplir con los requisitos definidos en el caso de uso.

Como podemos ver los casos de uso permiten la reutilización de código para múltiples plataformas ya que es totalmente independiente de frameworks.

La arquitectura hexagonal relacionado con las dependencias, sólo permite una única dirección y es de fuera hacia dentro. Esto quiere decir que la capa de la lógica de negocio (más interna) no debe depender de capas externas.

Para finalizar la arquitectura hexagonal es una definición de buenas prácticas que ayudan tanto a quienes trabajan con kits de desarrollo de software como a los que no.

2.2 Arquitectura Cebolla

La arquitectura cebolla (The Onion Architecture) tiene los mismos objetivos que la arquitectura hexagonal pero esta última no define como estructurar una aplicación.

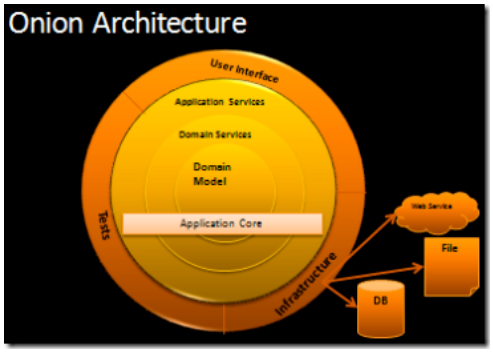

Figura 2. Arquitectura Cebolla

El termino *The Onion Architecture* fue creada por *Jeffrey Palermo* [2]. La principal característica de esta arquitectura como podemos ver en la Figura 2 es la de externalizar la base datos. Al externalizar la fuente de datos permite que la aplicación sea más fácil de mantener durante el periodo de vida de la misma ya que facilita la incorporación de tests al poder definir una única interfaz de acceso a datos independiente del modelo de datos que haya detrás. Sin embargo esta externalización supone un gran reto de diseño ya que generalmente se tiene la mentalidad de que una aplicación es una aplicación de datos y de que la lógica gira entorno a ella. Esto provoca que tanto la UI, la lógica de negocio y el acceso de datos estén muy acoplados, siendo imposible utilizarlas de forma separada y por lo tanto testearlas adecuadamente. Otra de las características de esta arquitectura es que la capa más interna contiene la lógica de negocio. Esta capa, al igual que en la arquitectura hexagonal, es independiente de las capas. El número de capas no está determinado y varía según las necesidades. Sin embargo, se tiene que cumplir la regla de dependencia (*Dependency inversion*) que indica que capas internas no deben conocer nada de las capas externas. A continuación

se detalla los elementos más comunes que se suelen encontrar en una aplicación diseñada teniendo en cuenta la arquitectura de cebolla:

- **Núcleo**: Contiene la lógica de negocio, servicios, interfaces e interfaces de repositorio.
- **UI**: Suele estar relacionada con frameworks y se implementan patrones como Model-View-Controller, Model-View-Presenter, Model-View-ViewModel, etc.
- **Test**: Pruebas unitarias o de integración.
- **Infraestructura**: Capa que implementa las interfaces del núcleo para tener acceso a datos, servicios, etc.

El objetivo de ambas arquitecturas es el mismo y consiste en que los proyectos que se realicen sean mantenibles y sin deuda técnica.

La mantenibilidad se puede definir como la ausencia o reducción de deuda técnica, es decir una aplicación será más fácil de mantener cuanto más lenta crezca la deuda técnica. La mantenibilidad tiene un enfoque a largo plazo ya que cuando se comienza un proyecto software es fácil de modelar porque no se han tomado hasta el momento decisiones. Pero cuando va avanzando el tiempo se agregan características que pueden entrar en conflicto con funcionalidades anteriores. Este problema se puede reducir utilizando un diseño de arquitectura que fomente las siguientes medidas:

- Realizar un cambio en un área de nuestro proyecto debería afectar a las otras lo menos posible.
- Añadir nuevas funcionalidades no deberían realizar grandes cambios en el núcleo de nuestro proyecto.
- El testing debería ser fácil de realizar.

Estas arquitecturas o buenas prácticas ayudan a reducir las malas decisiones. Si se toman malas decisiones al final se termina por hacer soluciones temporales que crean más problemas, por lo que definir una buena base arquitectónica reduce el crecimiento de la deuda técnica.

3. DEFINICIÓN DEL PROYECTO

3.1 Requerimientos funcionales

Los requisitos funcionales son las acciones fundamentales a las que dar respuesta el software. Los requisitos funcionales se dividirán respecto a dos roles bien definidos en la empresa Iluminación FM, son *empleado* y *administrador*.

- **RF1:** Requisitos funcionales para los administradores.

  A continuación se detalla los requisitos funcionales por parte de usuarios registrados en el sistema como *administradores*.

  - **RF1.1** Listado de usuarios.
    El usuario con rol de administrador cuando ejecute la aplicación obtendrá inmediatamente el listado de los usuarios con su información básica (nombre apellidos y correo), dividido en dos secciones (según el rol) y ordenados por fecha de alta.

- **RF1.2:** Creación de usuarios.

  El administrador podrá dar de alta a usuarios de rol administrador o empleado. Tras crearlos en el sistema se enviará al correo que haya introducido el administrador la contraseña temporal también introducida por el administrador que haya dado de alta al usuario.

- **RF1.3:** Eliminación de usuarios.

  Debe ofrece la posibilidad de dar de baja a usuarios. Se eliminará del sistema y recibirá una notificación a todos los dispositivos móviles donde esté registrado con esas credenciales.

- **RF1.4:** Listado de Partes de Localización Técnica.

  Este listado estará dividido en dos secciones, la primera identificará documentos no leídos por el usuario y la segunda por documentos ya leídos. Las dos secciones estarán ordenadas por la fecha de creación siendo el primer elemento el más reciente.

  En este listado encontraremos información básica del documento como por ejemplo la fecha de rodaje, la productora los días totales de rodaje y el responsable de la producción con su nombre, email y teléfono de contacto.

- **RF1.5:** Información completa del Parte de Localización Técnica.

  Si el usuario con rol de administrador pulsa uno de los documentos de la lista, se le mostrará toda la información del documento en otra vista, dividida en tres secciones debido al gran número de campos que compone el documento, como podemos ver en el **ANEXO I**. Si consulta uno de los documentos no vistos este cambiará de estado y será reclasificado a *leído*. Si se vuelve a la interfaz de listado de documentos se actualizará la lista de documentos sin necesidad de realizar una petición al servidor.

- **RF1.6:** Eliminación del Parte de Localización Técnica.

  En la interfaz donde se mostrará la información completa del documento se podrá también eliminar. Al regresar al listado de documentos este se actualizará sin necesidad de realizar una petición al servidor.

- **RF1.7:** Notificación de creación de Partes de Localización Técnica.

  Cuando un *empleado* envía al sistema un Parte de Localización Técnica se notifica a todos los *administradores* ya registrados en la aplicación mediante una ***notificación push*** y además se envía a sus correos el Parte de Localización Técnica en formato pdf.

- **RF2:** Requisitos funcionales para los empleados.

  Los usuarios registrados en el sistema como *empleados* tienen como requisitos funcionales:

  - **RF2.1:** Creación de Partes de Localización Técnica.

    El usuario podrá crear Partes de Localización Técnica siempre y cuando complete varios campos que son obligatorios que son:

    - **Fecha:** Fecha de creación del Parte de Localización Técnica.

- **Cliente:** Nombre de la empresa o persona que solicita los servicios de Iluminación FM.
- **Productora:** Nombre de la productora.
- **Producción:** Nombre de la producción.
- **Fecha de rodaje:** Día, mes y año del rodaje.
- **Total días de rodaje:** Número total de días de rodaje.
- **Equipo técnico necesario:**
  - **Carga:** Día, mes y año para la carga de material, número de personal y transporte necesario.
  - **Rodaje:** Día, mes y año, número de personal y transporte necesario para la realización del rodaje.
  - **Devolución:** Día, mes y año, número de personal y transporte necesario para la devolución.
- **Responsable de producción:** Nombre del responsable de producción.
- **Mail de contacto:** Mail de contacto del responsable de producción.
- **Best Boy/ Jefe elec:** Responsable del equipo de eléctricos, encargados de colocar los elementos de iluminación donde dice el *gaffer*.
- **Gaffer:** Técnico que selecciona los equipos convenientes para llevar a cabo la iluminación.
- **Key Grip:** Técnico encargado de los sistemas de agarre de fotografía, cine y video.
- **Localización:** Localización geográfica del proyecto. Cuando el *empleado* comience a escribir una ubicación el sistema le dará un listado de predicción junto con la opción de su ubicación actual.

- **RF3:** Requisitos comunes para administrador y empleado.

  Respecto a la aplicación Android Iluminación FM App tanto los administradores como los empleados comparten requisitos funcionales y son los siguientes:

  - **RF3.1:** Registro en múltiples dispositivos.

    Cualquier usuario podrá sincronizar su cuenta en todos los dispositivos móviles que quiera.

  - **RF3.2:** Notificación de baja en el sistema.

    Ya sea administrador o empleado podrá recibir una notificación de baja del sistema por parte de otro administrador a todos los dispositivos móviles donde esté registrado con esas credenciales.
    Si en el momento de recibir la notificación está con la aplicación en primer plano se le mostrará una pop-up y se eliminará sus credenciales y se le dirigirá a la vista de inicio de sesión. Otro de los casos que se puede producir es si no tiene la aplicación en primer plano se le notificará y se eliminarán sus credenciales.

  - **RF3.3:** Finalización de proceso de alta.

    La primera vez que inicia sesión un usuario o solicita recuperación de contraseña se le obligará a cambiar la contraseña temporal creada por el administrador que le haya dado de alta en el sistema.

  - **RF3.4:** Cambio de contraseña.

El usuario podrá cambiar su contraseña. Si se cambia de contraseña el sistema enviará una notificación de cambio de contraseña en todos los dispositivos donde haya iniciado sesión.

- **RF3.5:** Recuperación de contraseña.

  Si el usuario ha olvidado la contraseña, se le enviará a su correo una nueva contraseña.

- **RF4:** Requisitos Servidor.
  - **RF4.1:** Envío de notificaciones push.

    Encargado de enviar las notificaciones al servicio Google Cloud Messaging de Google [3][4].
  - **RF4.2:** Envío de datos a través de correo GMAIL.

    Se gestionará en el envío de datos por correo tras dar de alta usuarios, recuperación de contraseña y creación de documentos [5].
  - **RF4.3:** Renderizado de Parte de Localización Técnica a formato pdf.

    Cuando se recibe en el sistema un nuevo parte de localización Técnica se genera un pdf como la plantilla del **ANEXO I** añadiendo la información recibida [6].

3.2 Requerimientos no funcionales

Los requisitos no funcionales son los requisitos de interfaz, usabilidad, operacionales, de documentación, de rendimiento, de fiabilidad, seguridad, etc.
En la misma vista donde se encuentra el listado de usuarios se podrá navegar con un desplazamiento lateral hacia la sección de documentos creados por los *empleados*.

Los requisitos no funcionales son:

- **RNF1:** La aplicación está orientada a dispositivos táctiles con el sistema operativo Android a partir de la versión 4.0 (Ice Cream Sandwich).
- **RNF2:** La aplicación funcionará solo si está conectada a internet.
- **RNF3:** La seguridad está garantizada gracias al cifrado de los datos del usuario y en las peticiones al *backend*.
- **RNF4:** Utilización de *colas de mensaje* para mejorar el rendimiento en el backend.
- **RNF5:** Existirá mantenimiento de la aplicación tras las nuevas publicaciones de versiones del sistema operativo Android.
- **RNF6:** Diseño adaptable a los distintos tamaños de pantalla y densidad.
- **RNF7:** Control de errores para garantizar la estabilidad de la aplicación.
- **RNF8:** Caché de peticiones para mejorar el tiempo de respuesta.
- **RNF9:** La aplicación no necesita estar en segundo plano gracias al envío de notificaciones [3][4].
- **RNF10**: Las interfaces de creación (*empleado*) y visualización (*administrador*) del Parte de Localización Técnica estará dividido en tres pestañas debido al gran número de campos mejorando la usabilidad.

- **RNF11**: Aspecto *material design* en la aplicación incluso para la versión 4.0 de Android (Ice Cream Sandwich).

4. Diseño de la solución

Para la aplicación Iluminación FM se ha estudiado y realizado un diseño que recoge todos los requerimientos descritos en el Apartado 3.

4.1 Análisis de las Tecnologías

Según International Data Corporation (IDC) [7], Android es el sistema operativo para *smartphones* más extenso del mundo que alcanzó en Agosto de 2015 una cuota de mercados de más del 80% (véase Figura 3). En una primera versión Iluminación FM ha solicitado solo el desarrollo para el sistema operativo Android ya que a los empleados se les ha suministrado un dispositivo con este sistema operativo.

Period	Android	iOS	Windows Phone	BlackBerry OS	Others
2015Q2	82.8%	13.9%	2.6%	0.3%	0.4%

Figura 3. Cuota de mercado SO Smartphones 2015.

La tecnología empleada en esta versión de la aplicación de *Iluminación Fm App,* es la versión 4.0, con nombre *Ice Cream Sandwich.* Esta versión es la novena versión de este sistema operativo desarrollado por Google. Su objetivo fue desarrollar un sistema operativo para smartphones como para tablets, uniendo lo mejor de las versiones anteriores *Gingerbread 2.3* (smartphones) y *Honeycomb* (tablets).

4.1.1 ANDROID SDK

Android SDK [8] (Software Development Kit) incluye un conjunto de herramientas de desarrollo y bibliotecas API para crear, probar y depurar aplicaciones para Android. Viene incluido en el IDE oficial de Android llamada *Android Studio.*

4.1.2 ANDROID STUDIO

Android Studio es el IDE oficial para el desarrollo de aplicaciones basado sobre *IntelliJ IDEA* de *JetBrains* [9] y publicado de forma gratuita a través de la Licencia Apache 2.0. para Microsoft Windows, Mac OS X y GNU/Linux.

Como características a destacar tiene el renderizado de las vistas (*XML*) en tiempo real. Incluye una consola de desarrollador y la construcción de las aplicaciones están basadas en *Gradle.* Permite realizar *refactoring* de código de una forma rápida y además contiene herramientas *Lint* [10] para detectar problemas de rendimiento, usabilidad, compatibilidad de versiones, etc. Incluye plantillas para crear diseños de forma ágil junto a módulos como *Android Wear*, *Google Cloud Message*, *Android TV*, entre otros.

4.1.3 Ruby on Rails

Ruby on Rails es la tecnología utilizada para la parte del servidor y se ha utilizado en forma de API Rest para la comunicación entre el cliente y el servidor utilizando http

REquest y http Response. Ruby on Rails es la unión entre un lenguaje de programación (Ruby) y un framework (Rails). Es un entorno de desarrollo web, de código abierto que permite construir aplicaciones web flexibles y robustas. Está basado en el patrón de diseño Modelo Vista Controlador (MVC) pero para este proyecto no ha hecho falta seguirlo ya que el controlador no devuelve vistas sino respuestas en formato *JSON*.
Se optó desarrollar con el framework Ruby on Rails porque permite desarrollar aplicaciones escribiendo menos código y reduciendo la configuración, lo que supone reducir las horas de desarrollo y por lo tanto el coste. Además en el mundo laboral es sinónimo de calidad, ya que en proyectos *start up* se valora el uso de esta tecnología.

4.1.4 MongoDB

MongoDB es un sistema de bases de datos *NoSQL* orientado a documentos, es decir, en vez de guardar los datos en registros y tablas como hace en las base de datos relacionales, MongoDB guarda los datos con una estructura de tipo JSON binario con un esquema dinámico (formato *BSON*). Se ha utilizado un sistema de base de datos *NoSQL* y en concreto MongoDB porque permite realizar consultas geoespaciales y de texto mientras que las bases de datos relacionales no y como uno de los requisitos a la hora de realizar un Parte de Localización Técnica es el envío de la localización del rodaje se ha optado por esta tecnología. Esto permitirá en un trabajo futuro comparar trabajos anteriores realizados próximos a nuevos rodajes. Además, en las bases de datos *NoSQL,* no hay esquema de las migraciones ya que es el propio código quien define el esquema de la base de datos. Otro de los motivos por el que hemos seleccionado esta tecnología es puramente académico incorporando nuevos conocimientos sobre bases de datos *NoSQL*, que no son abordados en los contenidos del grado y máster.

Para entender los conceptos de una base de datos *NoSQL*a continuación se describen los elementos comunes:

SQL	MONGODB (NoSQL)
Tabla	Colección
Fila	Documento
Columna	Campo
Joins	Referencia de documentos

Para mapear las filas (documentos) de las tablas (colecciones) he utilizado *MongoId* (ODM) para MongoDB en Ruby.

4.1.5 RabbitMQ

RabbitMQ es un gestor de colas de mensajes para programación concurrente. Las colas de mensaje son un intermediario que facilita la comunicación entre emisores (publicadores) y receptores (consumidores). Sus principales características son:

- **Escalabilidad:** Permiten añadir más unidades de procesamiento lo que hace que nuestro sistema sea escalable. El propio sistema de colas se encargará de equilibrar la carga entre las diferentes unidades de procesamiento de forma transparente para consumidores y publicadores.
- **Asíncrona:** Dependiendo del funcionamiento de nuestro sistema o de nuestras necesidades queremos que los mensajes se vayan acumulando para procesarlos en lotes y por lo tanto la cola irá acumulando los mensajes hasta que se decida el momento de procesarlos.
- **Garantía de envío y orden:** Se garantiza que los mensajes lleguen al receptor en el orden en el que se han publicado.
- **Redundancia:** Si se produce un fallo en el sistema mientras se procesa un mensaje no hay que preocuparse de la pérdida de este, ya que esta estrategia permite a la cola persistir el mensaje hasta que sea procesado correctamente.
- **Flexibilidad:** Tiene una capa intermedia de comunicación entre procesos que permite desacoplarlos de la arquitectura del proyecto siempre y cuando cumplan los requerimientos de interfaz que representa la cola.
- **Elasticidad:** Si se produce situaciones como llegar al máximo de capacidad de recepción de peticiones y por lo tanto el sistema se ve incapaz de responder, las colas de mensajes permite equilibrar, filtrar y normalizar el flujo evitando el colapso del sistema.

Para la utilización de los servicios de RabbitMQ se ha utilizado una gema (cliente) en Ruby llamada Bunny que nos permitirá enviar y gestionar los mensajes de la cola.

4.2 Arquitectura Aplicación Android Iluminación FM

Como hemos visto en apartados anteriores el objetivo es desarrollar un software de calidad y como ya sabemos es difícil y complejo para que sea robusto, fácil de mantener, comprobable y lo suficiente flexible como para adaptarse al crecimiento y cambio. Se puede llegar a una solución válida siguiendo el siguiente conjunto de prácticas [11]:

- **Independiente de Frameworks:** La arquitectura no depende de la existencia de ninguna herramienta de desarrollo de software (SDK). Esto permite utilizar estos frameworks como herramientas, en lugar de meter sus limitaciones en nuestro sistema. Para esta primera versión del proyecto se ha desarrollado para Android lo que supone utilizar su SDK y se ha conseguido acotarlo de tal forma que la lógica de negocio no tenga constancia de su existencia.
- **Comprobable:** Las reglas de negocio pueden ser probadas sin la interfaz de usuario, base de datos, servidor web o cualquier otro elemento externo.
- **Independiente de la interfaz de usuario:** La interfaz de usuario puede cambiar constantemente sin tener que cambiar el resto del sistema, por ejemplo la interfaz de usuario de la aplicación Android Iluminación FM App se podría sustituir por una interfaz de usuario para una aplicación de escritorio para Windows, sin cambiar las reglas de negocio.

- **Independiente de la base de datos:** Puedes intercambiar Oracle o SQL Server con MongoDB, CouchDB ya que las reglas de negocio no está vinculados a la base de datos.
- **Independiente de agentes externos:** Las reglas de negocio no saben nada sobre el mundo exterior.

Teniendo en cuenta estas prácticas se diseña una arquitectura limpia por capas siguiendo la arquitectura cebolla junto a buenas prácticas de la arquitectura hexagonal (véase Figura 4). En la Figura 4 se detallan 4 capas, pero esto no quiere decir que sean obligatorias, ya que sólo son esquemáticas. Pero la *Regla de Dependencia* se aplica siempre, que como se puede ver las dependencias apuntan hacia el interior aumentando el nivel de abstracción.

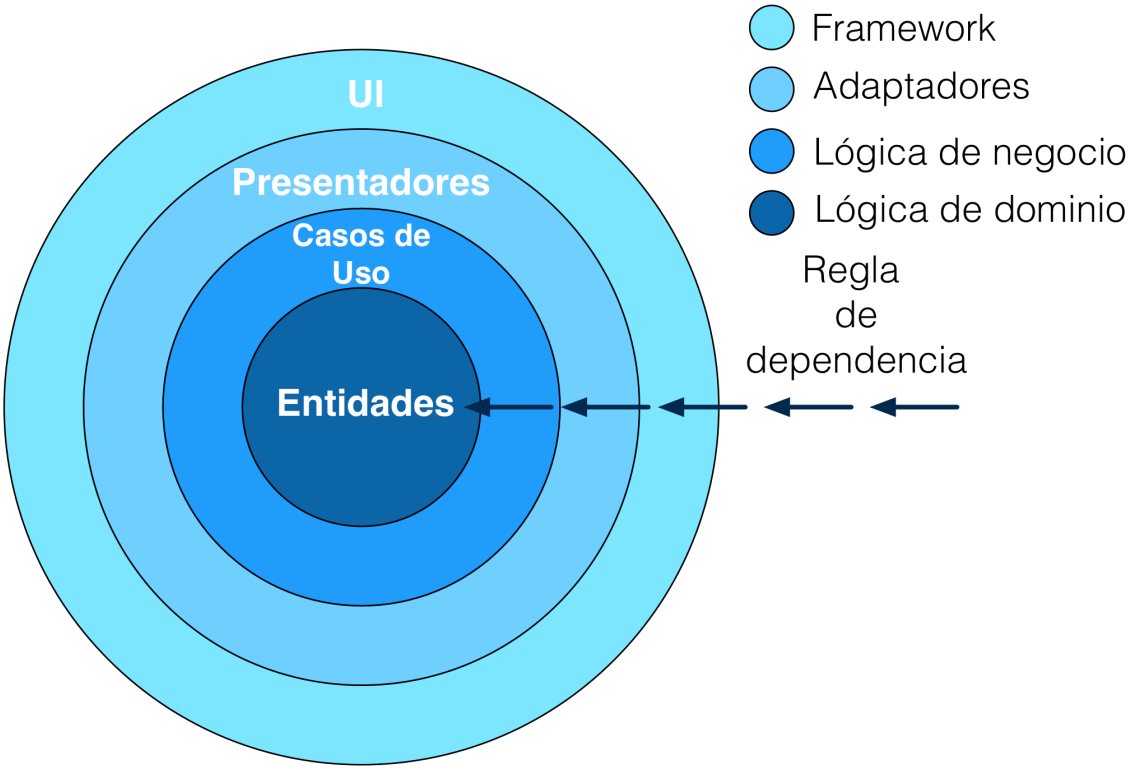

Figura 4. Arquitectura propuesta.

Antes de comenzar a diseñar la arquitectura Android para la aplicación Iluminación App hay que familiarizarse y comprender los siguientes términos:

- **Regla de dependencia:** Como se puede ver en la Figura 4 los círculos concéntricos representan áreas de software, los círculos externos son mecánicos y los internos encontramos las condiciones. Esta regla dice que ningún círculo interno puede saber nada acerca de círculos más externos a él, es decir, en el círculo interno no se debe mencionar ninguna función, clase, variable o cualquier otra entidad de software perteneciente a los círculos externos. Además los formatos de los datos en un círculo exterior no debe ser utilizado por un círculo interno, especialmente si esos formatos se generan por un *framework* de un círculo exterior.
- **Entidades:** Son los objetos o conjuntos de estructuras de datos y funciones utilizados en la lógica de negocio de la aplicación. Son los menos propensos a cambiar cuando hay cambios externos, como por ejemplo cuando hay un cambio

de navegación (IU). Por lo tanto ningún cambio operacional en la aplicación debería afectar a la capa de entidad.

- **Casos de Uso:** También denominados *interactors*, son los encargados de dirigir el flujo de datos hacia y desde las entidades, utilizando las reglas de negocio para alcanzar los objetivos del caso de uso.
- **Adaptadores de interfaz:** Encargados de convertir los datos en el formato más conveniente para los casos de uso, entidades y agentes externos como bases de datos o la Web. A este grupo pertenecen los presentadores y controladores.
- **Frameworks and Drivers:** Es la capa más externa y se compone de Frameworks y drivers como base de datos, frameworks de Web (ruby on rails), frameworks de smartphones (Android), etc.

El objetivo principal del diseño es mantener las reglas de negocio aisladas del mundo exterior, de tal modo que se puedan probar sin ninguna dependencia a cualquier elemento externo. Para este proyecto se ha dividido la aplicación Android en tres capas (véase Figura 5), donde cada una de ellas tiene un objetivo y trabaja de forma independiente del resto de bloques. Cada capa utiliza su propio modelo de datos, por lo que la transformación de ellos según la capa es un precio a pagar para no cruzar el uso de sus modelos en toda la aplicación. Para que se cumpla la regla de dependencia, la capa de dominio no debería depender de ninguna de las otras capas. La capa de datos debería depender sólo de la capa de dominio y no de la capa de presentación (framework). Y por último la capa de presentación debería depender de la capa de datos.

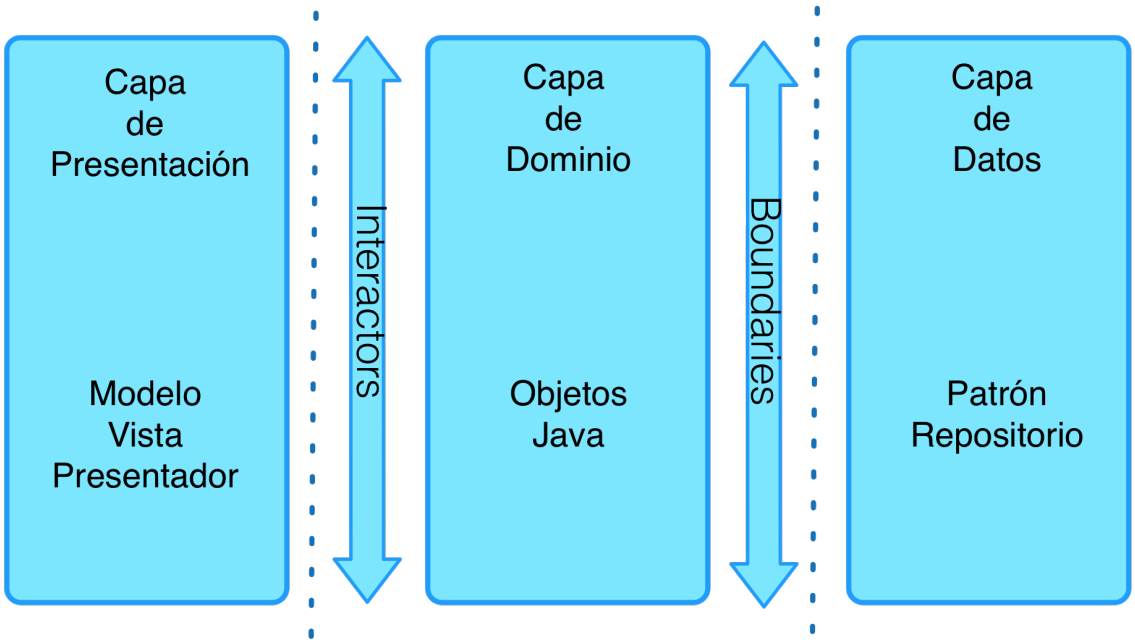

Figura 5. Estructura proyecto aplicación Iluminación FM Android.

4.2.1 Capa de Presentación

Esta capa está relacionada con las vistas, es decir, con elementos propios de Android, que son:

- **AndroidManifest.xml:** En Android, es un fichero de configuración de la aplicación que se va a desarrollar.

- **Action Bar:** El action bar de Android es la barra que aparece en la parte superior de la interfaz de la aplicación. Normalmente se muestra un icono, el título de la Actividad en la que se encuentra el usuario, botones de acción y un menú desplegable donde se muestra aquellos botones de acción que no tienen espacio.
- **Activity:** En Android, una Actividad es la encargada de crear la interfaz con la que el usuario va a interaccionar. La actividad tiene un ciclo de vida el cual hay que controlar.
- **Fragment:** En Android, un Fragment es una parte de la interfaz global de una pantalla. Esto permite modularizar mejor el código del programador y crear interfaces más complejas entre las pantallas grandes y pequeñas de los diversos dispositivos. El Fragment tiene que estar integrado en una Activity. Y al igual que la Activity tiene su propio ciclo de vida y es afectado por el ciclo de vida de la Activity.
- **FragmentActivity:** En Android, un FragmentActivity es una subclase de Activity que se introdujo en el paquete de compatibilidad *android-support.* En FragmentActivity se añadieron dos métodos para garantizar dicha compatibilidad con versiones anteriores a la 3 de Android.
- **View Group:** En Android, un View Group es una vista que puede contener otras vistas denominadas hijos.
- **ListView:** En Android, un ListView es una vista que muestra los ítems de la lista verticalmente.
- **ViewPager:** En Android, un ViewPager es una vista que permite al usuario desplazarse por las diferentes interfaces.
- **Adapter:** En Android, un Adapter actúa como un enlace entre la vista contenedora y los datos para esta vista. El Adapter proporciona acceso a los elementos de datos. También se encarga de hacer una vista por cada ítem del conjunto de datos.
- **Custom Adapter:** En Android, se puede customizar un Adapter cargando una vista diseñada al gusto del programador.
- **CustomView:** En Android, hay a veces que las vistas que proporcionan no cubren los requisitos que se le piden al programador. Además una custom view permite al programador crear vistas más robustas y reutilizables.
- **Intents:** Sirven para invocar componentes que en el contexto de Android serían actividades (UI), servicios (segundo plano) y proveedores de contenidos.

  Otro aspecto importante a la hora de utilizar custom views es la *retro compatibilidad* en una aplicación. Esto significa que la aplicación tendrá siempre el mismo aspecto para versiones anteriores.

Se utiliza un patrón de diseño Modelo Vista Presentador (MVP) para separar la interfaz de usuario de la lógica de la aplicación (véase Figura 6):

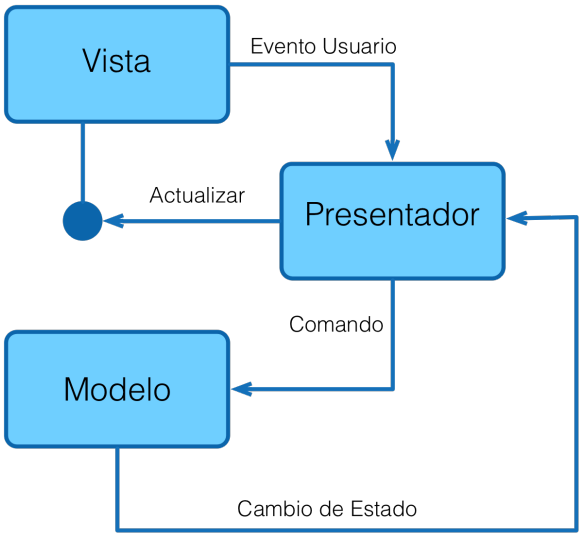

Figura 6. Patrón de Diseño Modelo Vista Presentador

- **Vista:** Compuesta de las ventanas y controles que forman la interfaz de usuario (*Activities y fragments)* y sólo contiene lógica de interfaz de usuario
- **Modelo:** Es donde se lleva a cabo toda la lógica de negocio.
- **Presentador:** Se compone de *interactors* (Casos de uso) y escucha los eventos que se producen en la vista y ejecuta los casos de uso en un nuevo hilo fuera del subproceso de interfaz de usuario de Android, y regresan usando una llamada de retorno con los datos que se solicitaron desde la vista. El presentador gestiona como se muestra la información en la vista, por lo tanto la vista no tiene ningún tipo de lógica, únicamente tiene como función el mostrar la información que se le pasa a través de la interfaz de la vista. A esta variación Martin Fowler la denomina *Vista Pasiva* [12].

4.2.1.1 Diseño de la interfaz

Para el diseño de la interfaz se ha procurado crear pantallas que respondan a todos los requisitos y que hagan que la aplicación sea intuitiva.

El diseño mostrado en la Figura 7 engloba toda la navegación permitida para el usuario ya sea administrador o empleado. Como podemos apreciar uno de los objetivos relacionados con la interfaz es haberle dado un aspecto *material design* (Android 5.0) sin haber utilizado la API correspondiente (API level 21) gracias a técnicas y librerías de compatibilidad.

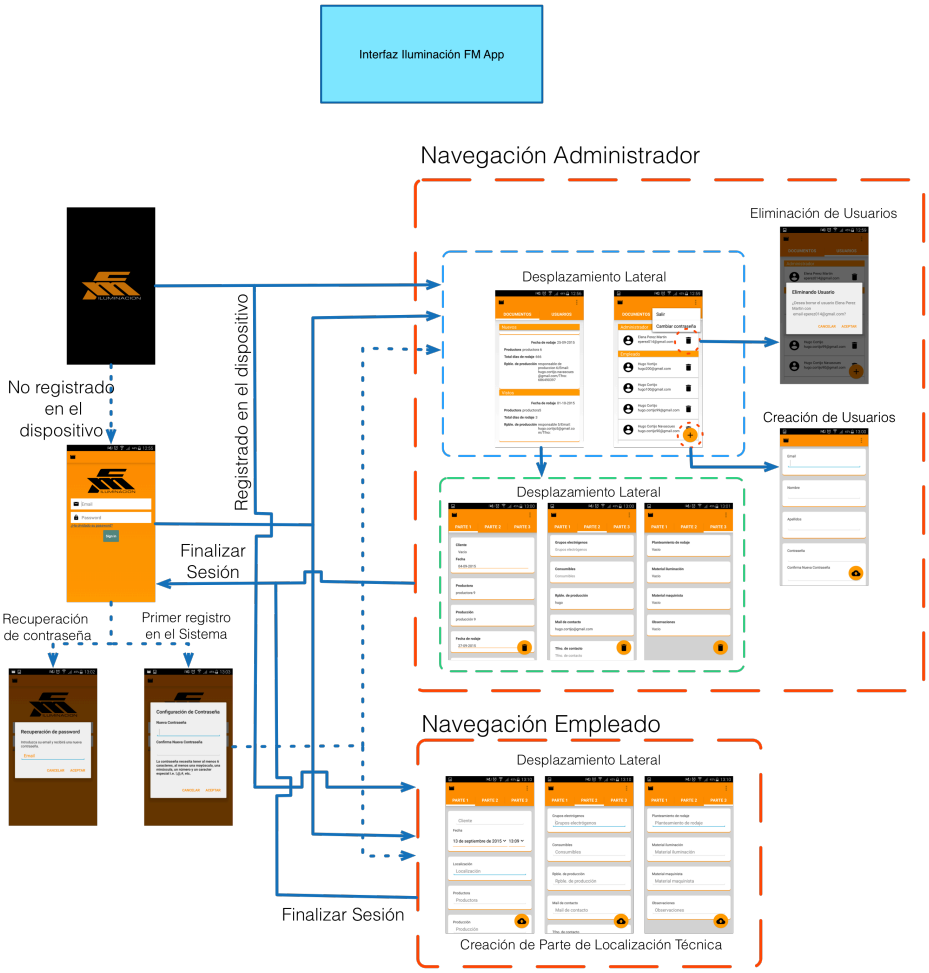

Figura 7. Interfaz de usuario Iluminación FM APP

La aplicación se inicia con una actividad con el logo corporativo, en ella en futuras mejoras podría iniciar procesos de actualización de información, por ejemplo guardar información de los empleados de la empresa para autocompletar los formularios de forma más rápida y que entre todos guarden el mismo patrón.

Tras finalizar la animación de la actividad de inicio si hay credenciales guardadas en la aplicación y dependiendo del rol de usuario se le mostrará su correspondiente navegación (Administrador o Empleado). Si no hubiera registro se navegaría hacia la pantalla de registro (véase Figura 7).

En la actividad de registro el usuario si se registra por primera vez en el sistema, la aplicación le solicitará que cambie su contraseña, ya que para haberle dado de alta, un administrador tuvo que asignarle un password. Una vez confirmado el cambio de contraseña o el registro es correcto dependiendo del rol del usuario se le dirigirá a una de las dos navegaciones:

- **Navegación Administrador:** La actividad principal para el usuario con rol de administrador está dividida en dos pestañas por las que podrá navegar desplazándose lateralmente.

  La primera pestaña contiene un listado de Partes de Localización Técnica divididos en dos secciones *Nuevos* y *Vistos* (véase Figura 7). Si se pulsa uno de los elementos de la lista, el usuario se dirigirá a la actividad donde se muestra toda la información del Parte de Localización Técnica. Respecto a la actividad que muestra el Parte de Localización Técnica se divide en tres pestañas para mejorar la estructuración de la información mostrada. El usuario sólo con deslizar el dedo o pulsando las pestañas podrá consultar los tres *fragments* en los

cuales en la parte inferior derecha aparecerá un botón que permitirá eliminar el Parte de Localización Técnica en el sistema.

En la segunda sección de la actividad principal del usuario con rol administrador, encontramos un listado de usuarios divididos en *Administradores* y *Empleados*. Además cada uno de los usuarios mostrados en el listado tiene su correspondiente icono de eliminación. A esta interfaz se le incorpora un botón situado en la parte inferior derecha de la pantalla, cuya función es iniciar un proceso de creación de usuario. Tras pulsar el botón de añadir un nuevo usuario se le mostrará al usuario una nueva pantalla donde tendrá que rellenar un formulario de alta. En esta pantalla siguiendo las prácticas de *material design* en la parte inferior derecha aparecerá un botón para registrar el nuevo usuario en el sistema.

En relación a la gestión de notificaciones cuando la sesión es iniciada por un usuario con rol administrador es notificar al usuario una vez que se registra o si se ha creado un nuevo Parte de Localización Técnica como podemos ver en la Figura 8:

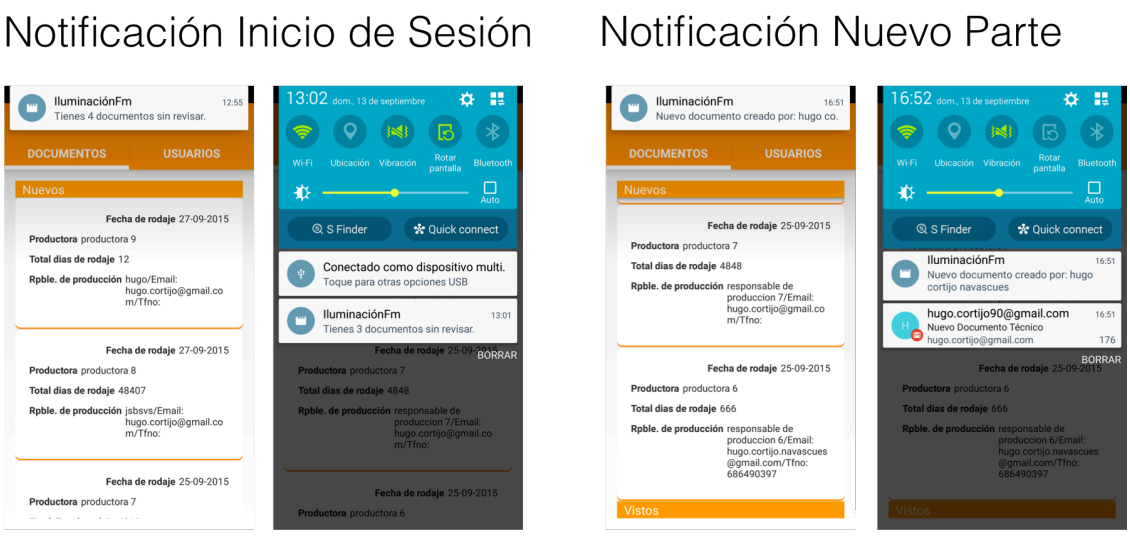

Figura 8. Notificaciones Administrador.

Navegación Empleado: En esta primera versión el rol de empleado solo tiene la actividad dividida en tres fragmentos para realizar Partes de Localización Técnica. Mencionar que para un trabajo futuro se añadirá otro tipo de formulario llamado *Parte de Personal* incluido en el **Anexo II**, que da una mejor visión del personal y material necesario para los distintos servicios que ofrece la empresa Iluminación FM. Sigue al igual que el resto de actividades prácticas propias de *material design,* incluyendo el botón que lanzará el evento de creación del nuevo Parte de Localización Técnica en el sistema en el lado inferior derecha de la pantalla.

- **Navegación común:** En relación a interfaces comunes a las dos navegaciones encontramos el menú desplegable con las opciones "*Cambiar Contraseña*" y "*Salir*" (véase Figura 9). Dicho menú está presente en todas las pantallas una vez que el usuario se ha registrado.

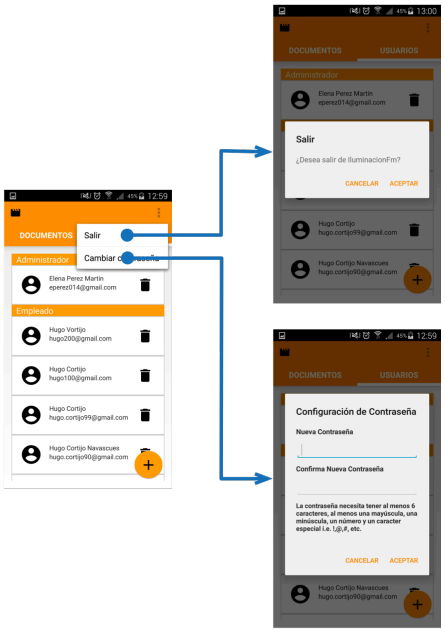

Figura 9. Menú Común.

Si se desea cerrar la sesión la aplicación dirige al usuario a la actividad de inicio de sesión (véase Figura 7). El cambio de contraseña se realizará sin redirigir al usuario a ninguna otra actividad.

Respecto a dar de baja a usuarios quien recibe la baja ya sea administrador o empleado la interfaz es la misma. A continuación se muestra la baja de un administrador cuando tiene la aplicación en primer plano y cuando la tiene cerrada o en segundo plano (véase Figura 10):

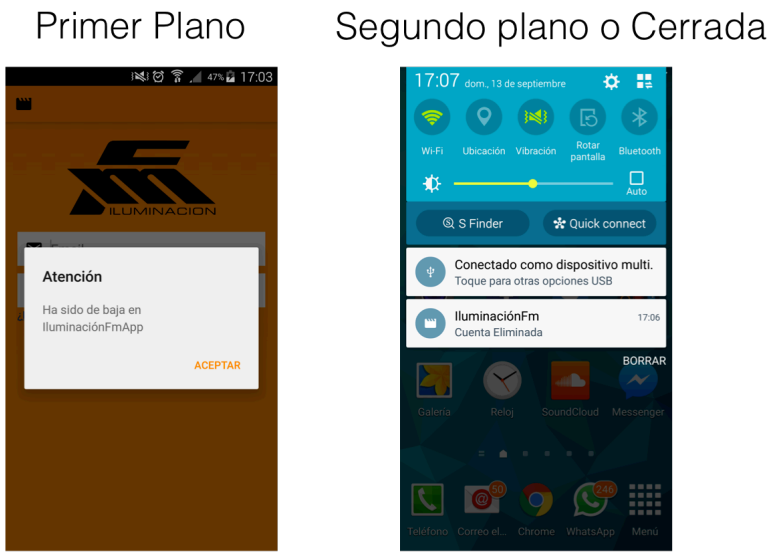

Figura 10. Captura de notificaciones push.

Si el usuario que recibe la notificación y tiene la aplicación abierta en primer plano se le dirige a la pantalla de inicio de sesión y se le muestra un pop-up notificándole que se le ha dado de baja. Si en el momento de recibir la baja no está trabajando con la aplicación se le mostrará una notificación en la bandeja de notificaciones del sistema operativo Android. Si pulsa dicha notificación se le abrirá la aplicación dirigiéndole a la pantalla de inicio de sesión.

4.2.2 Capa de Dominio

Este bloque contiene toda la lógica de negocio del proyecto. Es un módulo Java puro sin ninguna dependencia con el framework Android y es donde los *interactors* (Casos de uso) se encuentran implementados. Los componentes externos a este módulo utilizan interfaces cuando se conecta a los objetos propios de la lógica de negocio (véase Figura 11).

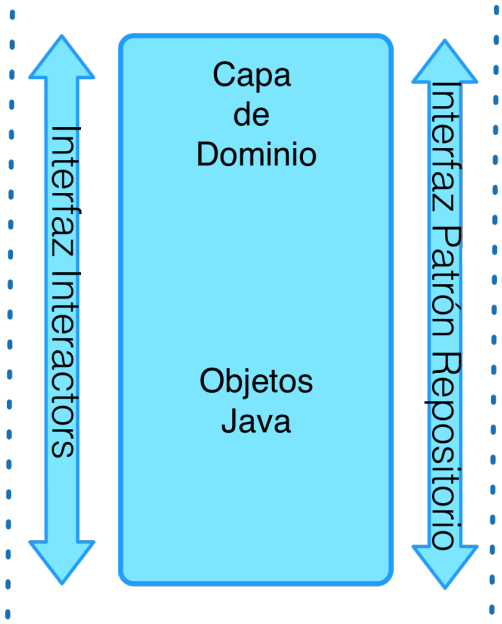

Figura 11. Capa de Lógica de Negocio

4.2.3 Capa de Datos

Todos los datos necesarios para la aplicación proceden de esta capa a través de una implementación de una interfaz que llamaremos *Repository* perteneciente a la capa de dominio que utiliza el patrón denominado *Repository* [13] aísla el acceso a diferentes fuentes de datos. Como podemos ver en la (véase Figura 12), la implementación del patrón *Repository* obtendrá los datos a través de distintas opciones (caché, base de datos, nube, etc) pero quien solicita los datos no sabrá el origen de estos.
En esta primera versión se implementará la recuperación de datos a través de la nube, ya que para recuperar los Partes de Localización Técnica se utilizará una caché de peticiones ya implementada, llamada *RoboSpice*. Gracias a RoboSpice, que incluye Retrofit, se consulta primero su caché para ver si contiene dicha petición y en caso de no encontrarla realiza la petición mediante Retrofit y la guarda en caché. Todo este proceso es transparente al desarrollador.

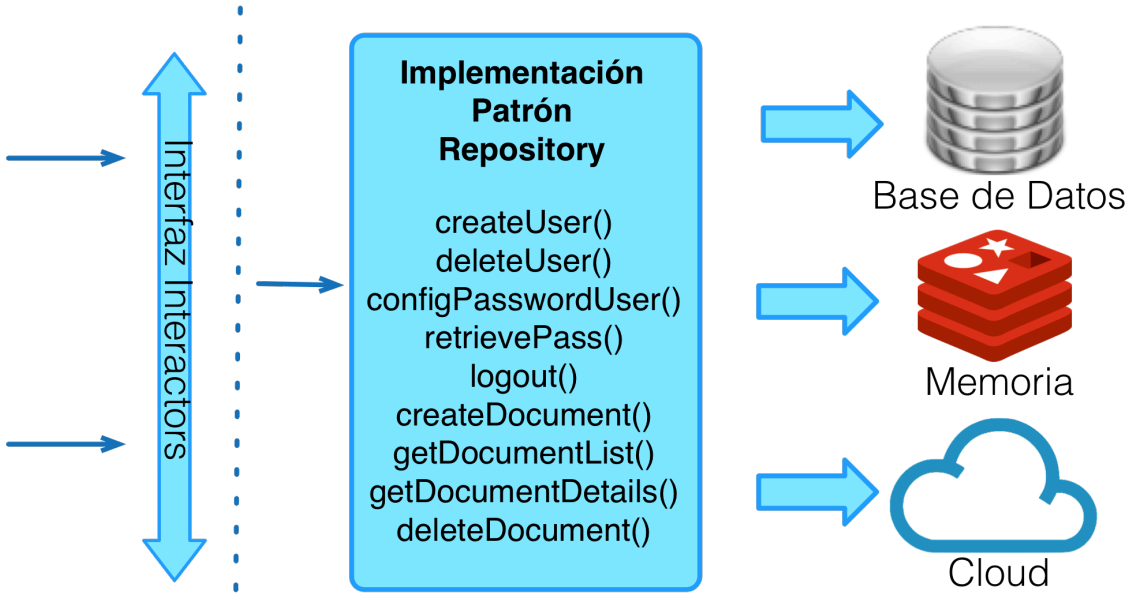

Figura 12. Capa de Datos.

4.2.4 Flujo entre capas

En esta sección veremos el flujo de llamadas para establecer la conexión entre los tres bloques que conforman la arquitectura de la aplicación Android Este flujo se describirá en mayor profundidad en la sección 5.1.5 con un ejemplo desarrollado en la aplicación Android.

En la Figura 13 se ha representado el flujo de llamadas genérico. El flujo comienza con la interfaz de usuario donde éste interacciona a través de un botón de refresco. Esta interacción provoca un evento que solicita un dato al presentador. El presentador ejecuta el caso de uso (*interactor*) correspondiente al contexto de la aplicación, que establece una conexión con la capa de datos a través de una interfaz. Una vez que la capa de datos adquiere el dato (o si se produce un error), se crea un evento que se transmite de vuelta por las diferentes capas hasta volver al presentador quien, finalmente, controla la actualizaciones en la interfaz del usuario.

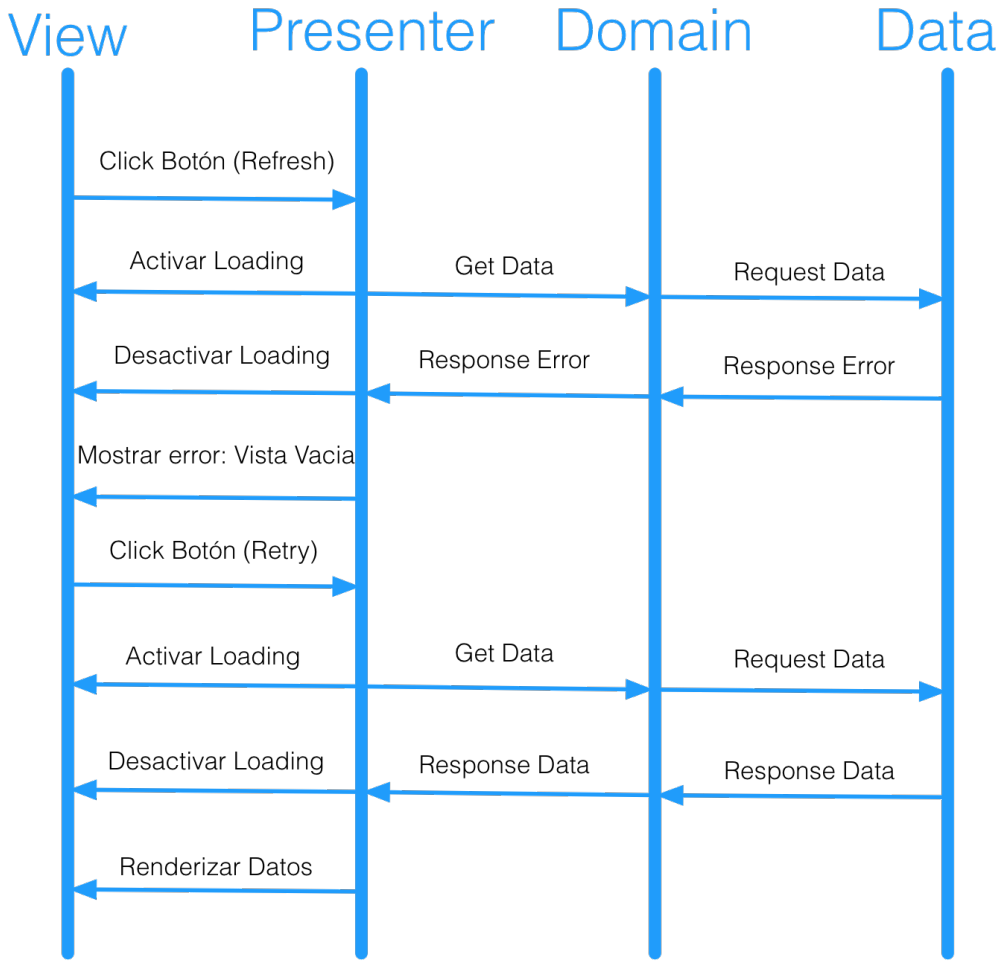

Figura 13. Flujo de llamadas App Android

La comunicación entre las diferentes capas se hace a través de *Listeners* (View-Presenter) y *callbacks* (Presenter-Domain-Data) que consiste en implementar interfaces declaradas por el desarrollador en la capa del domain para que el presentador recoja los datos y en la capa data para que la capa domain recoja datos. Este enfoque trae algunas dificultades porque tener un gran número de cadenas de devolución de llamadas (*callbacks*) podría comprometer la legibilidad del código. Otra alternativa que podría suplantar a los *Listeners* es la utilización de un sistema de bus de eventos, como por ejemplo *Otto* [14], pero utilizar este tipo de solución es como usar *GOTO* y en una programación estructurada debe evitarse [15].

4.3 API Rest

La API Rest es la parte del sistema que se encarga de gestionar las peticiones de la parte cliente. Dicha API está desarrollada con el framework Ruby on Rails como hemos visto en la sección 4.1.3. La estructura generada para el proyecto Iluminación FM API Rest es la siguiente:

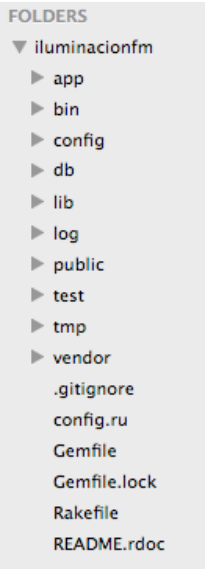

Figura 14. Estructura proyecto Iluminación FM en Ruby on Rails

Los directorios o ficheros más importantes son:

- **App/:** Es el núcleo de la aplicación, incluye modelos, vistas, controladores y ayudas. En esta primera versión solo utilizaremos los directorios:
  - **Assets/:** Recursos compartidos por la aplicación como imágenes, javascript, css.
  - **Controllers/:** Este directorio como podemos ver en la Figura 15 se ha incluido un nuevo directorio llamado api donde encontramos los controladores de la API Rest.

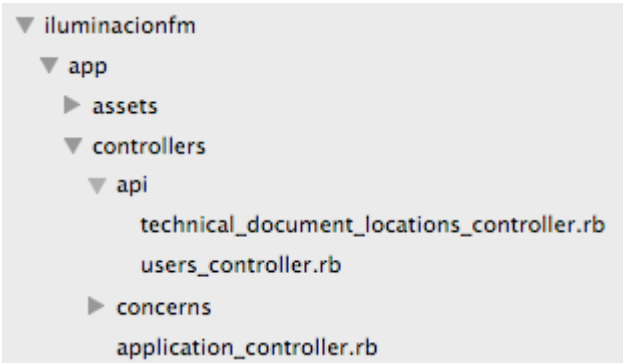

Figura 15. Directorio App.

El controlador *application_controller.rb* es autogenerado y se le ha otorgado el objetivo de validar las peticiones, es decir quien envía la petición es un cliente válido para utilizar dicha API.

  - **Models/:** Contiene la lógica de negocio de la API.
  - **View/:** Esta parte incluye únicamente la implementación para transformar a formato pdf una plantilla hecha en html para los Partes de Localización Técnica.
- **Config/:** Contiene la configuración de la aplicación. Contiene ficheros para la internacionalización de la aplicación, variables de entorno en modo de desarrollo, test o producción. También se configura la conexión a la base de datos y las rutas registradas que dará soporte a los clientes.
- **Db/:** En dicho directorio se ubican scripts en ruby para manipular la base de datos.

- **Lib/:** Módulos de biblioteca desarrollados por el desarrollador, en este proyecto se ha incluido un módulo de cifrado y descifrado simétrico. Además se ha incorporado un directorio que contiene scripts de indexación para la base de datos MongoDB que veremos en la sección 5.
- **Log/:** Contiene los logs para entornos en desarrollo, test o producción.
- **.gitignore:** Fichero que contiene patrones para que el control de versiones *Git* ignore.
- **Gemfile:** Fichero que contiene las librerías (*gemas*) utilizadas para la aplicación.

4.4 Diseño de la base de datos

A continuación se muestra el diseño de la base de datos a la que accede el servidor para gestionar los usuarios y los partes de localización técnica.

4.4.1 Diseño Preliminar

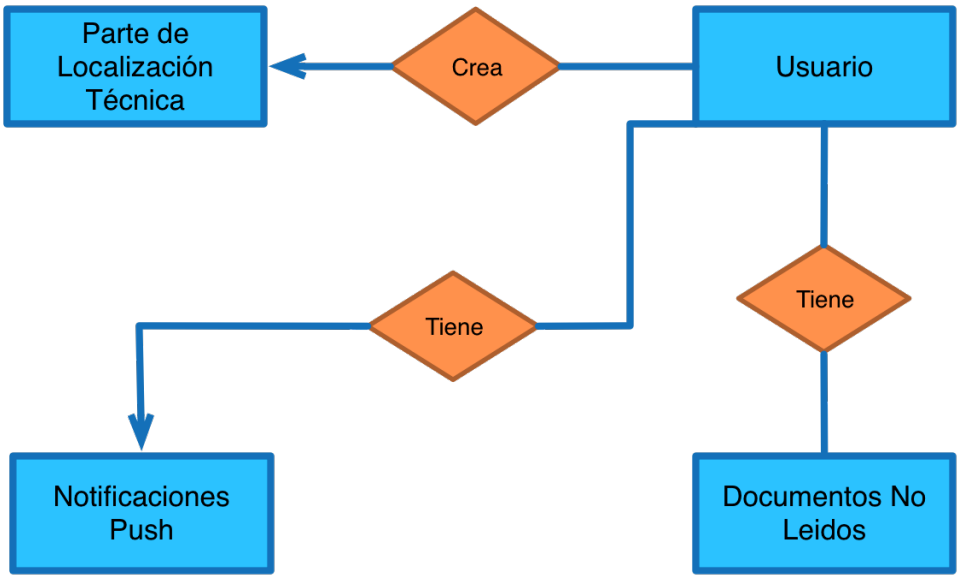

Figura 16. Entidad Relación Iluminación FM App

En esta primera versión solo se ha considerado tener las cuatro entidades mostradas en la Figura 16. A continuación se mostrará el diseño final detallando los campos de cada entidad.

4.4.2 Diseño Final

Como podemos ver en la Figura 17 la base de datos está formada por las siguientes *colecciones*:

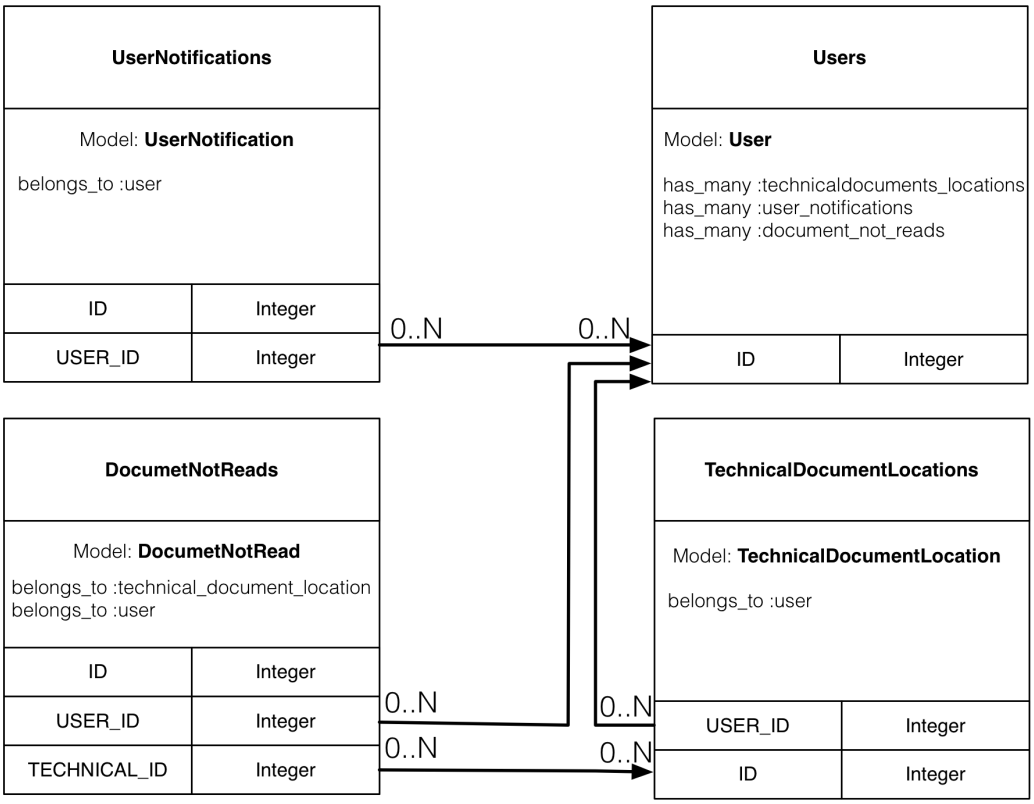

Figura 17. Diseño Final Base de Datos.

- **Users:** Los campos que contiene esta entidad son:
  - confirmPass: Este campo de tipo *boolean* sirve para detectar si es la primera vez que se registra un usuario en el sistema, es decir, cuando le ha dado de alta un administrador y accede a través de la aplicación Android con la contraseña enviada a su correo.
  - email: Contiene el correo electrónico del usuario para poder notificarle a través del correo de todos los cambios en el sistema (nuevos documentos, recuperación de contraseña y alta en el sistema).
  - name: Nombre del usuario.
  - secondName: Apellidos del usuario.
  - password: Contraseña del usuario. No se guarda la contraseña en plano del usuario sino que se le aplica una función hash y se cifra con un algoritmo criptográfico simétrico (véase sección 4.6). Este campo junto a *email* se utilizan para validar todas las peticiones.
  - role: Este campo determinará los permisos en cuanto a las consultas y funcionalidades específicas. Se diferencian dos roles que son administrador y empleado.

- **TechnicalDocumentLocations:**
  - date_created: Este campo se crea automáticamente cuando se añade en cualquier documento en su respectiva colección. Es de tipo Time y nos permitirá reordenar la lista de Partes de localización Técnica en la aplicación móvil si se visualiza un parte no visto hasta el momento o se elimina.
  - client: De tipo String y guarda el nombre del cliente.
  - film_producer: De tipo String y contiene el nombre de la productora.
  - production: De tipo String y contiene el nombre de la producción.
  - filming_date: Fecha de rodaje. Campo de tipo Time.

- total_filming_days: Número total de días de rodaje. Campo de tipo Integer.
- director_photography: Director de fotografía del proyecto.
- technical_equipment_nec: Descripción resumida del equipo técnico necesario para realizar el proyecto.
- freight: Fecha (dd/mm/yyyy) de carga del material en el transporte. Campo de tipo Time.
- filming: Fecha (dd/mm/yyyy) del montaje del material de iluminación. Campo de tipo Time.
- devolution: Fecha (dd/mm/yyyy) de devolución del material en el almacén. Campo de tipo Time.
- transport: Número de transporte necesario para llevar el material.
- gensets: Campo de tipo String y sirve para especificar los kilowatios necesarios.
- consumables: Sirve para especificar si necesitan filtros o gelatinas especiales. Campo de tipo String.
- respon_production: Responsable de la producción.
- mail_contact: Email de contacto del responsable de producción.
- tfno_contact: Telefono del responsable de producción.
- gaffer: Nombre del técnico que selecciona los equipos convenientes para llevar a cabo la iluminación.
- best_boy: Nombre del responsable del equipo de eléctricos, encargados de colocar los elementos de iluminación donde dice el *gaffer*
- key_grip: Nombre del técnico encargado de los sistemas de agarre de fotografía, cine y video.
- special needs: Campo de tipo String para reflejar necesidades especiales del rodaje.
- approach_filming: Planteamiento del rodajes, es decir, tiempo de rodaje.
- lighting_equipment: Lista del material básica.
- material_engineer: Lista de maquinaría básica.
- observations: Observaciones relacionados con el rodaje como por ejemplo problemas de acceso a la zona de rodaje.
- location: Campo de tipo Array de dos posiciones para guardar la localización del proyecto (altitud y latitud)
- accurancy: Observaciones respecto
- user_id: Referencia de tipo ObjectId al usuario que ha enviado el Parte de Localización Técnica.

- **UserNotifications:** Almacena los dispositivos a los que están asociados a un usuario y los campos de estos documentos son:
  - deviceId: Identificador de tipo String único obtenido del dispositivo móvil.
  - pushId: Token de tipo String obtenido a través del servicio Google Cloud Message para el envio de notificaciones.
  - user_id: Referencia de tipo ObjectId al usuario que tiene registrado un dispositivo.
- **DocumentNotReads:** Contiene documentos que relacionan usuarios que no han consultado a través de la aplicación los Partes de Localización Técnica. Los campos que se almacenan son:

- technical_document_location: Campo de tipo ObjectId que referencia al documento almacenado en la colección TechnicalDocumentLocations.
- user_id: Referencia de tipo ObjectId al usuario que no ha visualizado el Parte de Localización Técnica en la aplicación móvil.

4.5 Arquitectura del sistema

La arquitectura del sistema de Iluminación FM App también pertenece a este trabajo de fin de master (véase Figura 18). Como hemos visto en secciones anteriores en esta primera versión el cliente estará desarrollado en Android. Respecto al bloque servidor hay que destacar que se utiliza el servicio de Amazon Elastic Compute Cloud (Amazon EC2) que permite un control completo de los recursos, escalar rápidamente según las necesidades, pagar solo por los recursos utilizados y además los primeros 12 meses son gratuitos mientras no se superen los recursos.
En este momento hay una instancia la cual se ha incorporado una API Rest en Ruby on Rails que da soporte a las peticiones enviadas desde el cliente. Además consulta la base de datos NoSQL llamada MongoDB y envía tareas que serán ejecutadas en paralelo gracias a la cola de mensajes RabbitMQ como por ejemplo renderizado de una plantilla html a pdf y envío de correos con dichos pdfs con gran carga de computación.

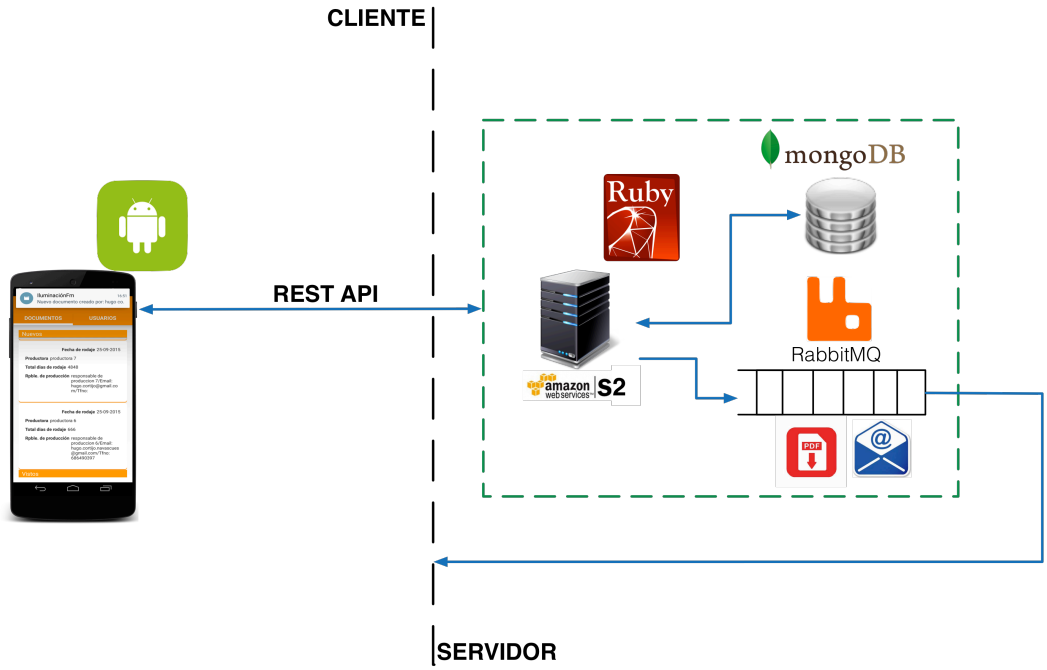

Figura 19. Arquitectura del Sistema Iluminación FM App

4.6 Seguridad

Debido al gran número de peticiones *HTTP* que involucran el id del usuario y por lo tanto toda su información habrá que desarrollar una capa de seguridad. Se utilizará una criptografía simétrica entre el dispositivo móvil del usuario con el *backend* para cifrar el id del usuario y del dispositivo asociado y no ser vulnerable a ataques. Para mantener la información a buen recaudo en la base de datos, las credenciales de los usuarios no se guardan en texto plano. Primero se aplica una función *hash* y luego se le aplica un cifrado simétrico. La clave simétrica que utiliza el cliente es diferente a la clave simétrica que utiliza el servidor para guardar las credenciales en la base de datos para

aumentar la complejidad en la obtención de información comprometida. El servidor conoce la clave simétrica utilizada en el dispositivo móvil. Se ha decidido cifrar los datos mediante *AES128*+Salt ya que hoy en día computacionalmente es imposible mediante fuerza bruta descifrar la información encriptada mediante este algoritmo. Además los tokens encriptados tendrán una fecha de caducidad imposibilitando un ataque *MitM*.

Además, como medida extra de seguridad todo usuario debe tener una contraseña con más de 5 caracteres, y que contenga al menos una mayúscula, una minúscula, un número y un carácter especial.

5. Solución

5.1 Aplicación Android

En esta sección describiremos la estructura a nivel de clases de la aplicación Android a través de diagramas de clase. Como he explicado anteriormente se divide en tres subproyectos (capas) en los cuales está dividido por paquetes. A continuación veremos con más detalle la implementación de los diferentes bloques.

5.1.1 Capa Presentación

En esta sección veremos cómo está estructurada la capa de presentación a través de sus paquetes y clases. También se detallará como se ha implementado el patrón de diseño MVP con un ejemplo de la aplicación, junto a técnicas de desarrollo y utilización de librerías que evitan volver a implementar soluciones estándard.

5.1.1.1 Descripción Estructural

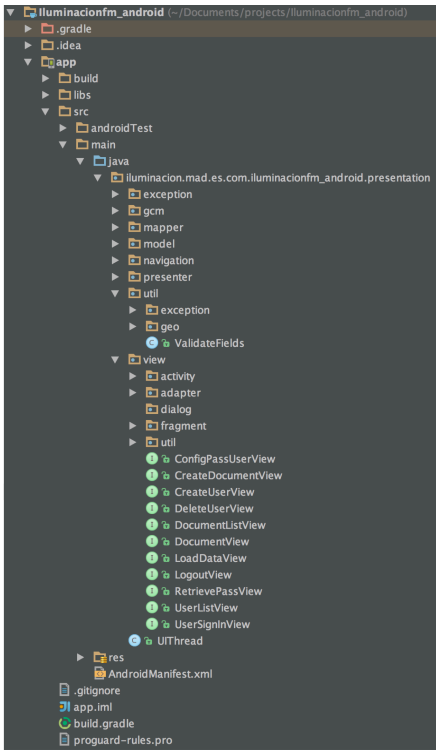

Figura 20. Estructura App (Android)

Para profundizar y entender el diseño arquitectónico propuesto a continuación veremos cómo está estructurada la capa de presentación en el proyecto Android denominada App (véase Figura 19).

- **Paquete Exception**: Es muy importante mantener el formato de los diferentes mensajes de error que pueden producirse en la aplicación. Por lo tanto en este paquete encontramos la clase *ErrorMessageFactory*, cuyo único método crea un mensaje de error dependiendo del tipo de instanciación de la excepción, pasada por argumento.

- **Paquete GCM**: Contiene las clases que gestionan las notificaciones enviadas desde el servicio Cloud Messaging de Google [3][4]. Consiste en un *IntentService* que se ejecuta en segundo plano completamente invisible para el usuario y cuando termina de procesar el mensaje recibido y mostrarlo a través de una notificación en el sistema operativo Android

termina su ejecución.

- **Paquete Mapper**: Este paquete contiene clases que se encargar de transformar el modelo de datos de la capa de dominio al modelo de datos de la capa de presentación (ver sección 4.2 para más detalles).
- **Paquete Model**: En este paquete se implementa el modelo de datos de la capa de presentación.
- **Paquete Navigation**: En el encontramos la clase *Navigator* encargada de obtener los *Intents* que invocarán a las diferentes actividades de la aplicación permitiendo al usuario navegar por la aplicación.

Paquete Util: En este paquete encontramos utilidades como recoger la ubicación actual del dispositivo si el usuario tiene activado la opción *Ubicación* en su dispositivo móvil. Además contiene una clase llamada *ValidateFields* que comprueba si los datos introducidos por el usuario son válidos. Si no fueran válidos existe un paquete exception, el cual contiene clases que crean excepciones. Un ejemplo sería lanzar una excepción de tipo *PasswordException* cuando la contraseña introducida no cumple varias premisas.

- **Paquete Presenter**: Contiene las clases que pertenecen al patrón de diseño MVP. Todas las clases implementan la interfaz *Presenter* y además hacen de punto de conexión con la capa de dominio ejecutando los casos de uso.
- **Paquete View**: Está compuesto de cuatro paquetes muy importantes a la hora de mostrar al usuario la interfaz gráfica, controlar los eventos recibidos y renderizar datos procedentes de la capa de dominio y son:
  - **Activity:** Este paquete contiene siete clases, *BaseActivity* extiende de *FragmentActivity* y el resto menos *IniActivity* (Splash) heredan de *BaseActivity*. La clase *InitActivity* es la pantalla de introducción de la aplicación y se encarga de mostrar el logo corporativo de la empresa introducido en una animación. Cuando termina la animación redirige a una de las tres actividades según si el usuario no se ha registrado (*LoginActivity*) o si se ha registrado con rol de administrador (*AdminMainActivity*) o empleado (*EmployeeMainActivity*).
    El resto de clases heredan de la clase *BaseActivity* ya que implementa dos presentadores que tienen que ser accesibles cuando cualquier usuario se ha registrado, y son poder cambiar su contraseña y salir de la aplicación.
    AdminMainActivity es la actividad principal para los usuarios con rol de administrador y está compuesta de dos *Fragments* (*UserListFragment* y *TechnicalDocumentListFragment*) contenidos en un *ViewPager* (véase).

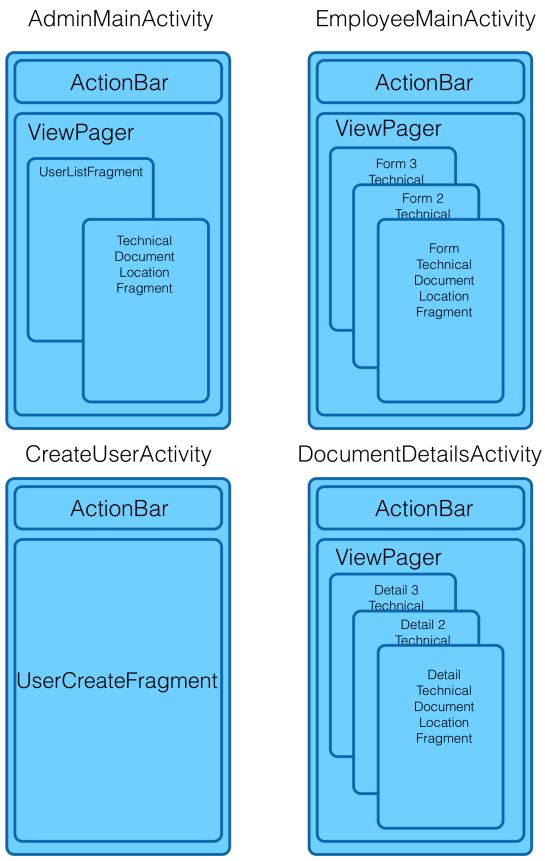

Figura 20. Estructura de las Actividades Principales.

La clase EmployeeMainActivity es la actividad principal del usuario con rol empleado y contiene tres *Fragmentos* en los cuales se ha dividido el Parte de Localización Técnica, (véase **Anexo I**) donde el empleado podrá completar el formulario.

La clase *DocumentDetailsActivity* estructuralmente es igual a *Employee-MainActivity* pero solo es una vista de consulta de los Partes de Localización Técnica creados por los empleados.

La actividad *LoginActivity* fue generada gracias al IDE Android Studio donde se le añadió los casos de uso pertinentes y modificando el diseño. Y para finalizar la clase *CreateUserActivity* que contiene un *Fragment* encargado de coger los datos de alta de un nuevo usuario (véase Figura 20).

- **Adapter:** En este paquete encontramos clases encargadas de crear vistas gracias a los datos obtenidos por el presentador a través de la capa de dominio. Para esta primera versión se ha utilizado para crear los ítems de las *ListViews* (*PinnedSectionListView*) de los Fragments UserListFragment y *Technical-DocumentLocationFragment* y también sirve para incorporar *Fragments* a un *ViewPager.*
- **Fragment:** El paquete fragment contiene diez *Fragmentos* de los cuales *BaseFragmentes* es una clase abstracta que hereda de *Fragment* y que tiene un método abstracto (*initializePresenter()*) para inicializar un presentador. El resto de fragmentos de la aplicación heredan de *BaseFragment* e sobrescriben el método *initializePresenter*.

- **Clase UIThread:** Esta clase es utilizada por parte de los casos de uso. Se encarga de notificar (*Handler*) al hilo principal (UI) de lo ejecutado en segundo plano por parte de otro hilo (véase Figura 21).

```
public interface PostExecutionThread {

    /**
     * Causes the {@link Runnable} to be added to the message queue of the Main UI Thread
     * of the application.
     *
     * @param runnable {@link Runnable} to be executed.
     */
    void post(Runnable runnable);
}
```

```
public class UIThread implements PostExecutionThread {

    private static class LazyHolder {
        private static final UIThread INSTANCE = new UIThread();
    }

    public static UIThread getInstance() {
        return LazyHolder.INSTANCE;
    }

    private final Handler handler;

    private UIThread() { this.handler = new Handler(Looper.getMainLooper()); }
    @Override
    public void post(Runnable runnable) { handler.post(runnable); }
}
```

Figura 21. Clase UIThread.

5.1.1.2 MVP

Para entender completamente el estudio estructural de la capa de presentación se han realizado un diagrama de clases (véase Figura 22) relacionado con la obtención de los Partes de Localización Técnica del sistema. Además complementará a como se ha desarrollada la solución de implementación del patrón de diseño MVP.

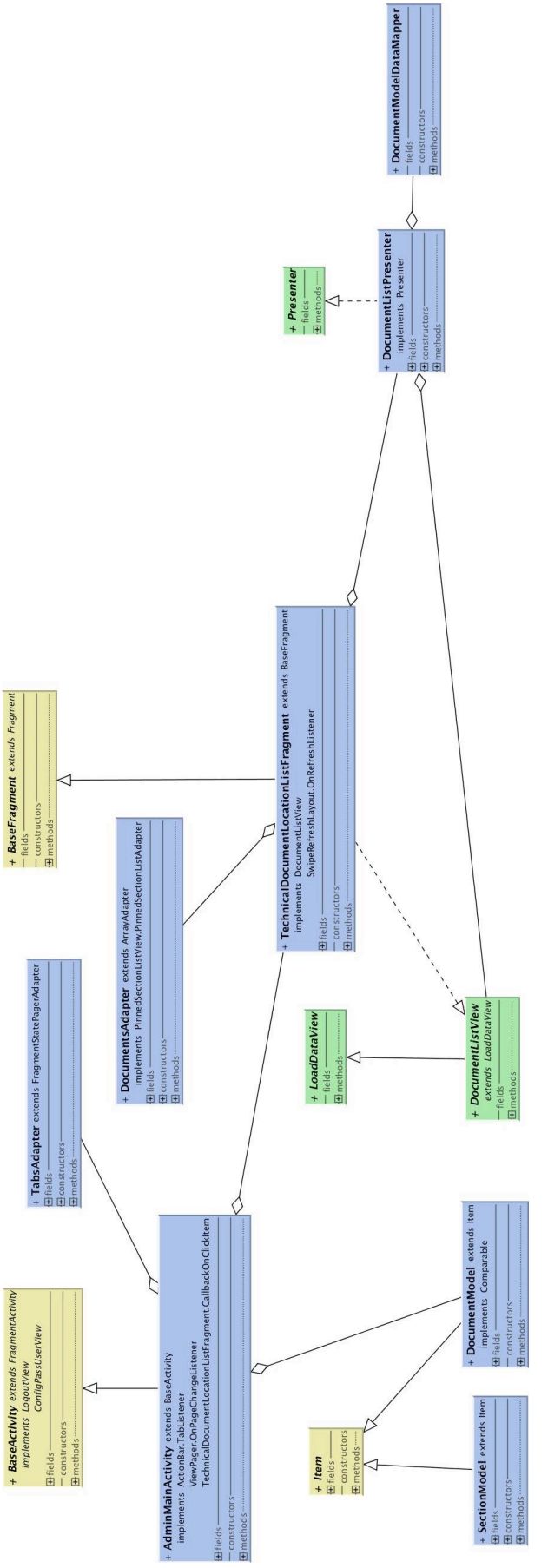

Figura 22. Diagrama de Clases Bloque App.

Como podemos ver en la Fihura 22 el fragmento *TechnicalDocumentLocationList-Fragment* implementa la vista *DocumentListView*. Esta vista es controlada por el presentador *DocumentListPresenter* inicializado en el propio fragmento sobrescribiendo el método *initializePresenter* (véase Figura 23).

```
@Override
void initializePresenter() {

    ThreadExecutor threadExecutor = JobExecutor.getInstance();
    PostExecutionThread postExecutionThread = UIThread.getInstance();

    ObjectEntityDataMapper documentEntityDataMapper = new ObjectEntityDataMapper();
    DataStoreFactory documentDataStoreFactory =
            new DataStoreFactory(this.getContext());
    Repository documentRepository = DataRepository.getInstance(documentDataStoreFactory,
            documentEntityDataMapper);
    DocumentModelDataMapper documentModelDataMapper = new DocumentModelDataMapper();
    this.itemModelDataMapper = new ItemModelDataMapper();

    GetDocumentListUseCase getDocumentListUseCase = new GetDocumentListUseCaseImpl(documentRepository,
            threadExecutor,postExecutionThread);
    this.documentListPresenter = new DocumentListPresenter(this, getDocumentListUseCase, documentModelDataMapper);


}
```

Figura 23. Inicialización del Presentador *DocumentListPresenter*

Una vez inicializado el presentador (veáse Figura 23), se le solicita desde el fragmento que ejecute el caso de uso (*this.documentListPresenter.initialize(...)*), controlando la lógica de la vista implementada en el Fragment, ya que el presentador tiene la instancia de la vista pasada en el primer argumento de su constructor (*new DocumentListPresenter(**this**, getDocumentListUseCase, documentModelDataMapper)*) como podemos ver en la Figura 24.

TechnicalDocumentLocationListFragment

```
private void loadDocumentsList(String beforeId, int limit) {
    if (this.documentListPresenter != null) {
        this.documentListPresenter.initialize(beforeId, limit);
    }
}
@Override
public void hideRetry() {
    this.retryView.setVisibility(View.GONE);
    this.notDocumentsView.setVisibility(View.GONE);

}

@Override
public void showLoading() {
    if(!swipeView.isRefreshing()) {
        mProgressView.setVisibility(View.VISIBLE);
    }
}
```

DocumentListPresenter

```
/**
 * Initializes the presenter by start retrieving the technical document location list.
 */
public void initialize(String beforeId, int limit) {
    this.beforeId = beforeId;
    this.limit = limit;
    this.loadDocumentsList();
}

/**
 * Loads all documents.
 */
private void loadDocumentsList() {
    this.hideViewRetry();
    this.showViewLoading();
    this.getDocumentList();
}

public void onDocumentClicked(DocumentModel documentModel) {
    this.documentListView.viewDocument(documentModel);
}

private void hideViewRetry() { this.documentListView.hideRetry(); }

private void showViewLoading() { this.documentListView.showLoading(); }

private void getDocumentList() {
    this.getDocumentListUseCase.execute(this.beforeId, this.limit,documentListCallback);
}
```

Figura 24. Implementación Vista y Presentador.

Tras solicitar los datos al modelo a través de la ejecución del caso de uso (*this.getDocumentListUseCase.execute(...)*) se recibirá una respuesta capturada en la implementación del *Callback documentListCallback* en el presentador *DocumentList-Presenter* (véase Figura 25).

```
private final GetDocumentListUseCase.Callback documentListCallback = new GetDocumentListUseCase.Callback() {

    @Override public void onDocumentListLoaded(Collection<Document> documentCollection) {
        DocumentListPresenter.this.showDocumentsCollectionInView(documentCollection);
        DocumentListPresenter.this.hideViewLoading();
    }

    @Override public void onError(ErrorBundle errorBundle) {
        DocumentListPresenter.this.hideViewLoading();
        DocumentListPresenter.this.showErrorMessage(errorBundle);
        DocumentListPresenter.this.showViewRetry();
    }
};
```

Figura 25. Implementación Callback en *DocumentListPresenter*.

La Interfaz *GetDocumentLisUseCase.Callback* permite la conexión entre la capa de dominio (modelo) con la de presentación, ya que se inicializa en el presentador y se pasa como argumento al caso de uso. Ahora en el caso de uso es donde se utilizan los métodos *onDocumentListLoaded* si ha obtenido Partes de Localización Técnica o en caso de error usar el método *onError*.

5.1.1.3 Técnicas de Desarrollo

- ***Técnica ViewHolder***

La Técnica *ViewHolder* surge debido a que cuando en un adaptador se van creando las vistas en el método *getView()* tenemos que utilizar una o varias veces la sentencia *findViewById()*. Esta sentencia es muy costosa y provocaba errores de memoria en la aplicación. Para resolver esto se incluyen los ID que identifican a cada elemento dentro de la propia View.

El objeto View tiene un método *getTag()* y *setTag()*, que permiten guardar información en la vista y en este caso evitar llamadas innecesarias a *findViewById()*. Como ejemplo se ha cogido el adaptador que crea las vistas de los Partes de Localización Técnica para insertar en una *ListView*. Simplemente hace falta crear una clase que en este caso se ha llamado Document*ViewHolder* y en el método *getView*() comprobar si la vista contiene los ID (véase Figura 26). Esta técnica se ha utilizado en todos los contenedores de vistas de la aplicación mejorando el rendimiento de esta.

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    DocumentViewHolder documentViewHolder = null;

    if (convertView == null || !(convertView.getTag() instanceof DocumentViewHolder)){

        if(items.get(position).getType() == Item.SECTION ){

            convertView = this.layoutInflater.inflate(R.layout.section_item, parent, false);
            documentViewHolder = new DocumentViewHolder();
            documentViewHolder.nameSection = (TextView) convertView
                    .findViewById(R.id.nameSection);
            convertView.setTag(documentViewHolder);


        }else{

            convertView = this.layoutInflater.inflate(R.layout.row_document, parent, false);
            documentViewHolder = new DocumentViewHolder();
            documentViewHolder.filmingDateView = (TextView) convertView.findViewById(R.id.value_filming_day);
            documentViewHolder.filmProducerView = (TextView) convertView.findViewById(R.id.value_film_producer);
            documentViewHolder.responsibleProducerView = (TextView) convertView.findViewById(R.id.value_responsible_production);
            documentViewHolder.totalDaysFilmingView = (TextView) convertView.findViewById(R.id.value_total_filming_days);
            convertView.setTag(documentViewHolder);

        }


    } else {
        documentViewHolder = (DocumentViewHolder) convertView.getTag();
    }
```

```
static class DocumentViewHolder {
    TextView filmingDateView;
    TextView filmProducerView;
    TextView totalDaysFilmingView;
    TextView responsibleProducerView;
    TextView nameSection;

}
```

Figura 24. Técnica ViewHolder.

5.1.1.4 Librerias

- ***Material Dialogs***

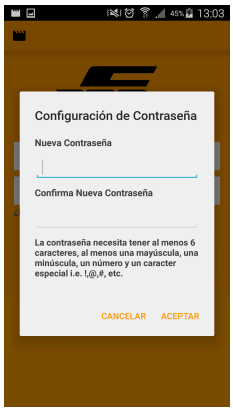

Figura27. MaterialDialog con una CustomView.

Material Dialogs es una librería de Aidan Folestad y se encuentra en el repositorio GitHub [16]. Uno de los objetivos relacionados con el diseño de la aplicación era darle un aspecto *material design,* sólo disponible a partir de la versión Android 5.0. Como la aplicación está orientada a dispositivos táctiles con el sistema operativo Android a partir de la versión 4.0 (Ice Cream Sandwich) se ha utilizado dicha librería para que los *pop-up* tengan aspecto *material design* (véase Figura 27).

- ***Pinned Section ListView***

Figurae 25. Seccion Nuevos se bloquea al hacer scroll.

Pinned Section ListView es una librería de Halfbit y se encuentra en el repositorio GitHub [17] y el comportamiento de esta librería se puede ver en YouTube [18].
Se ha utilizado esta librería para organizar los usuarios por el rol y los Partes de Localización Técnica consultados o no. Al recorrer la lista de ítems que pertenecen al mismo conjunto (si es por rol administrador y empleado o si es por partes de localización nuevos y vistos), la cabecera de la sección se quedará fija (véase Figura 28).

- ***DateTimePicker***

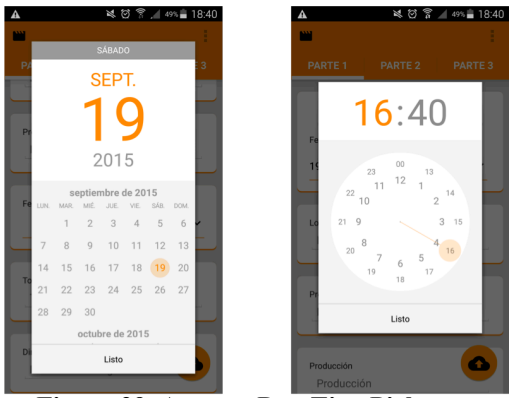

Figura 29. Aspecto DateTimePicker en IluminaciónFM.

DateTimePicker es una librería de CiTuX y se encuentra en el repositorio GitHub [19]. Siguiendo las pautas de *material design* a la hora de seleccionar la fecha y la hora se ha utilizado esta librería para varios de los campos a introducir en el Parte de Localización Técnica como podemos ver en la Figura 29.

- ***Google Cloud Messaging***

Google Cloud Messaging [3][4] es otra librería de Google que permite enviar datos desde un servidor a un dispositivo con Android y enviar mensajes entre dispositivos Android. En IluminaciónFM App se ha utilizado para notificar a los administradores de la creación de Partes de Localización Técnica y para la gestión de bajas del sistema y por lo tanto de la aplicación. Si el usuario tiene el móvil apagado cuando encienda el móvil los servidores del Google detectan que se ha encendido el dispositivo y si tiene notificaciones pendientes de recibir se le enviarán.

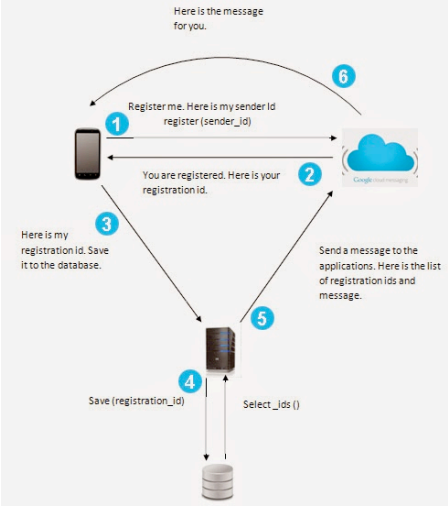

Figura 30. Ciclo de Vida de GCM.

Como se puede ver en la Figura 30, cuando el usuario se registra en IluminaciónFM App a continuación se registra el dispositivo (paso 1) y el servicio de Google le responde con un identificador. Ese identificador se envía en la petición correspondiente al registro (paso 3) y es almacenado en la base de datos de IluminaciónFM App terminando el proceso de registro del usuario (paso 4). Cuando se envía una notificación a un usuario a través del servidor de IluminaciónFM, se utiliza el servicio de Google enviando el mensaje junto al token del usuario (*resgistration_id* asociado al dispositivo) guardado en la base de datos (paso 5). El servicio de Google envía el mensaje cuando el dispositivo está encendido.

- ***Android View Animations***

Es una librería de Daimajia que encontramos en el repositorio GitHub [20] la cual aporta a vistas animaciones. Se ha utilizado para simular animaciones propias de *material design*, en concreto cuando se carga un Parte de Localización Técnica y se ha renderizado los datos en la vista además se muestra el botón de borrar con una animación.

5.1.2 Capa Dominio

En esta sección veremos cómo está estructurada la capa de dominio a través de sus paquetes y clases (véase Figura 31). También se detallará como se ha implementado el patrón estructural *Repository* con una estrategia que a través de una fábrica, recoge fuentes de datos. En esta versión inicial sólo está implementada la obtención de datos a través de la API Rest.

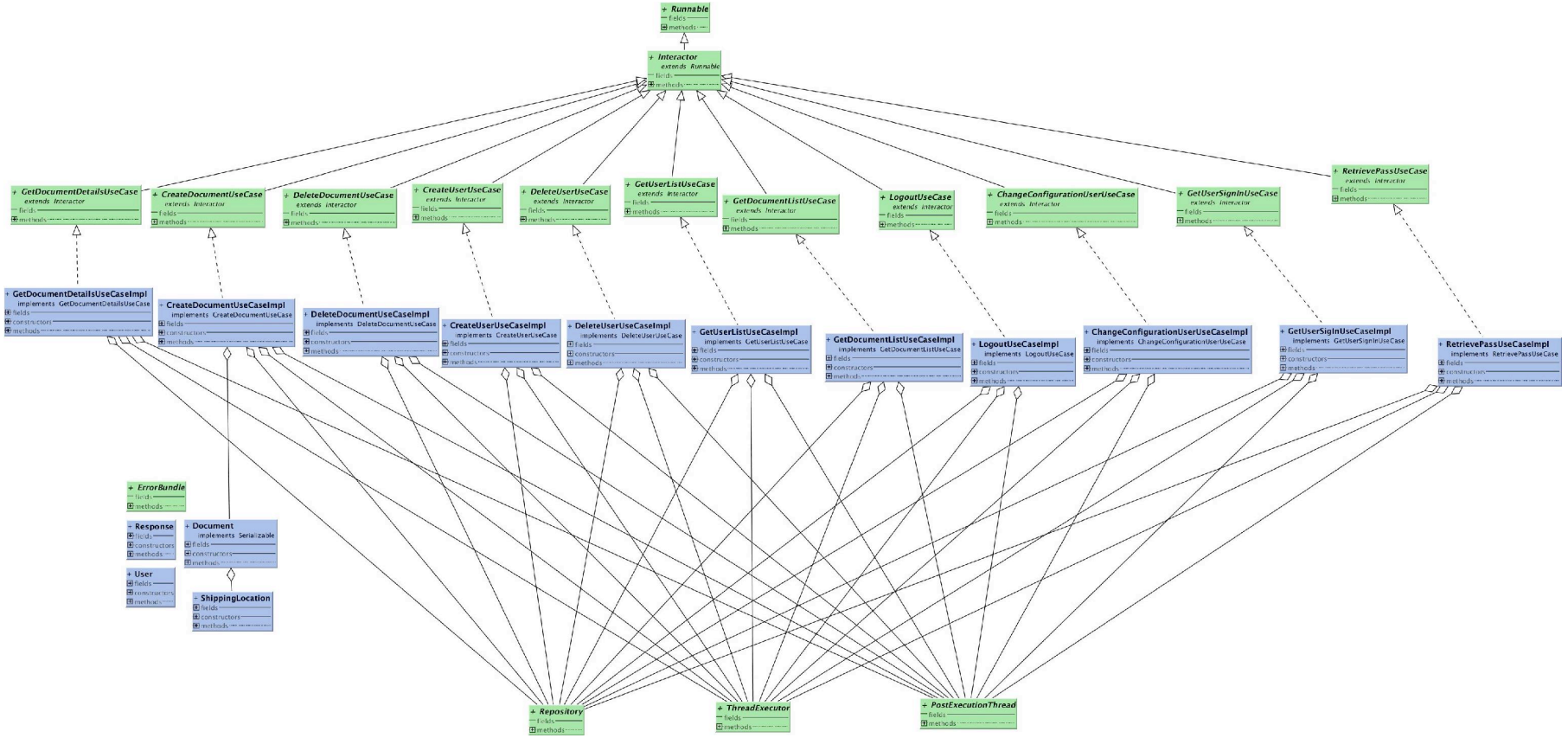

Figura 31. Diagrama de Clases Capa Domain.

5.1.2.1 Descripción Estructural

La capa de dominio como principal característica es que no tiene ninguna dependencia Android, es un módulo de Java puro. Está compuesto por los siguientes paquetes (véase Figura 32):

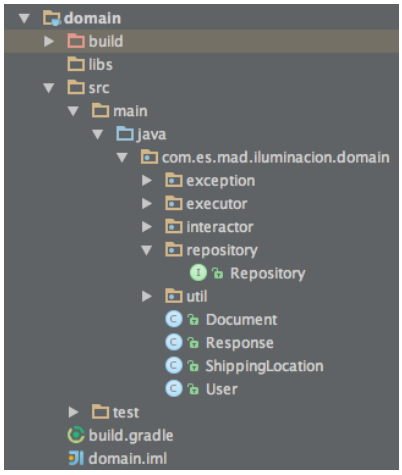

Figura 32. Estructura Domain.

- **Paquete Exception**: Contiene la interfaz *ErrorBundle* y sirve para recubrir las excepciones procedentes de la capa Data.
- **Paquete Executor**: En este paquete contiene dos interfaces muy importantes. Una de ellas es *PostExecutionThread* para abstraer el contexto de ejecución cuando se cambia de un hilo a otro. Es útil para encapsular el hilo de la interfaz (UI) ya que los casos de uso se harán en segundo plano por lo tanto es útil para cambiar el contexto y actualizar la interfaz de usuario. La otra interfaz llamada *ThreadExecutor* es la encargada de ejecutar los casos de uso (*interactors*) fuera del hilo de interfaz de usuario.

```
public interface Interactor extends Runnable {

    /**
     * Everything inside this method will be executed asynchronously.
     */
    void run();
}
public interface GetDocumentDetailsUseCase extends Interactor {

    /**
     * Callback used to be notified when either details document have been loaded or an error
     * happened.
     */
    interface Callback {
        void onDocumentDetailsLoaded(Document document);
        void onError(ErrorBundle errorBundle);
    }

    /**
     * Executes this use case.
     * @param documentId
     * @param callback A {@link GetDocumentDetailsUseCase.Callback} used to notify the client.
     */
    void execute(String documentId, Callback callback);

}
public class GetDocumentDetailsUseCaseImpl implements GetDocumentDetailsUseCase {

    private final Repository documentRepository;
    private final ThreadExecutor threadExecutor;
    private final PostExecutionThread postExecutionThread;

    private String documentId;
    private GetDocumentDetailsUseCase.Callback callback;


    public GetDocumentDetailsUseCaseImpl(Repository documentRepository, ThreadExecutor threadExecutor,
                                         PostExecutionThread postExecutionThread) {
        if (documentRepository == null || threadExecutor == null || postExecutionThread == null) {
            throw new IllegalArgumentException("Constructor parameters cannot be null!!!");
        }
        this.documentRepository = documentRepository;
        this.threadExecutor = threadExecutor;
        this.postExecutionThread = postExecutionThread;
    }

    @Override
    public void execute(String documentId, Callback callback) {

        if ( StringUtils.isNullOrEmpty(documentId)|| callback == null) {
            throw new IllegalArgumentException("Invalid parameter!!!");
        }
        this.documentId = documentId;
        this.callback = callback;
        this.threadExecutor.execute(this);
    }

    @Override
    public void run() {
        this.documentRepository.getDocumentDetails(this.documentId, this.repositoryCallback);
    }

    private final Repository.DocumentDetailsCallback repositoryCallback =
            new Repository.DocumentDetailsCallback() {
                @Override public void onDocumentDetailsLoaded(Document document) {
                    notifyGetDocumentDetailsSuccessfully(document);
                }

                @Override public void onError(ErrorBundle errorBundle) { notifyError(errorBundle); }
            };

    private void notifyGetDocumentDetailsSuccessfully(final Document document) {
        this.postExecutionThread.post(() -> {
            callback.onDocumentDetailsLoaded(document);
        });
    }

    private void notifyError(final ErrorBundle errorBundle) {
        this.postExecutionThread.post(() -> { callback.onError(errorBundle); });
    }
```

Figura 27. Interactor GetDocumentDetailsUseCaseImpl.

- **Paquete Interactor**: De este bloque el elemento más importante es la interfaz *Interactor*. Es una interfaz común heredada por todas las interfaces que se utilizan para los diferentes casos de uso. Esta interfaz representa una unidad de ejecución para los diferentes casos de uso y el resultado que deberá ser devuelto a través de un *callback* que debe ser ejecutado en el hilo de interfaz de usuario (véase Figura 33).

La implementación de este caso de uso (*GetDocumentDetailsUseCase-Impl*) como podemos ver implementa la interfaz *GetDocumentDetailsUse-Case*) que extiende de la interfaz *Interactor*. Como vimos en la Figura 24 el presentador llama al método *execute* del caso de uso y este ejecuta la implementación de dicho método gracias a la instancia *ThreadExecutor* cuya implementación ejecutará el *Runnable* (método *run()* de GetDocumentDetailsUseCaseImpl). Como podemos ver en el método *run* (ejecutado fuera del hilo de la UI) se le pasa un *Callback* que permitirá

cambiar de contexto entre el hilo de ejecución del *interactor* y el de la UI (*this.postExecution-Thread.post()*).

- **Paquete Repository**: Contiene la interfaz *Repository* que es implementada por la clase *DataRepository* perteneciente a la Capa Data, en la sección 5.1.2.2 se detallará la implementación.

5.1.2.2 Patrón Repository

El patrón *repository* es un patrón estructural que permite abstraer la implementación de acceso a datos en nuestra aplicación, provocando que nuestra lógica de negocio no sepa de la existencia de la fuente de datos. Por lo tanto este patrón actúa de intermediario entre la lógica de negocio y la lógica de acceso a los datos para centralizar todo en un mismo punto y no replicar código.

Como podemos ver en la Figura 34 en IluminaciónFM App la clase *DataRepository* solicita *DataStoreFactory* una fuente de datos a través del método *create()*. En el método *create()* en esta primera versión crea la fuente de datos *CloudDataStore* que obtendrá los datos a través de la API Rest solicitado en está en la fuente.

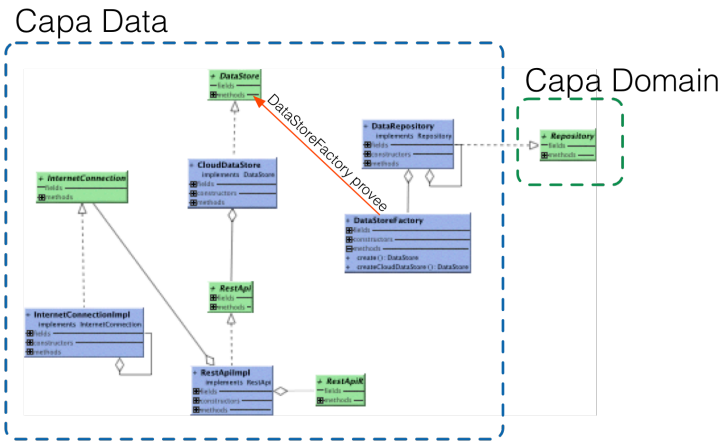

Figura 34. Patrón Repository en IlumninaciónFM APP.

Como trabajo futuro se añadirá nuevas fuentes de datos como acceso a base de datos local. Añadir una nueva fuente de datos se haría de una forma rápida sin necesidad de reestructurar el código implementado, simplemente añadiendo una nueva clase (fuente) de acceso a datos y lógica en el método *create()* de *DataStoreFactory* comprobando si el dato solicitado está en la nueva fuente.

5.1.3 Capa Datos

Todos los datos provienen de la capa de datos que implementa la interfaz *Repository* (Capa Domain) que utiliza el patrón Repository como hemos visto en la sección 5.1.2.2. En esta sección veremos cómo está estructurada la capa de datos a través de sus paquetes y clases (véase Figura 35), cómo se ha evitado el uso de *AsyncTask* gracias a la clase *JobExecutor* y la utilización de las librerías *Retrofit* [21] y *RoboSpice* [22].

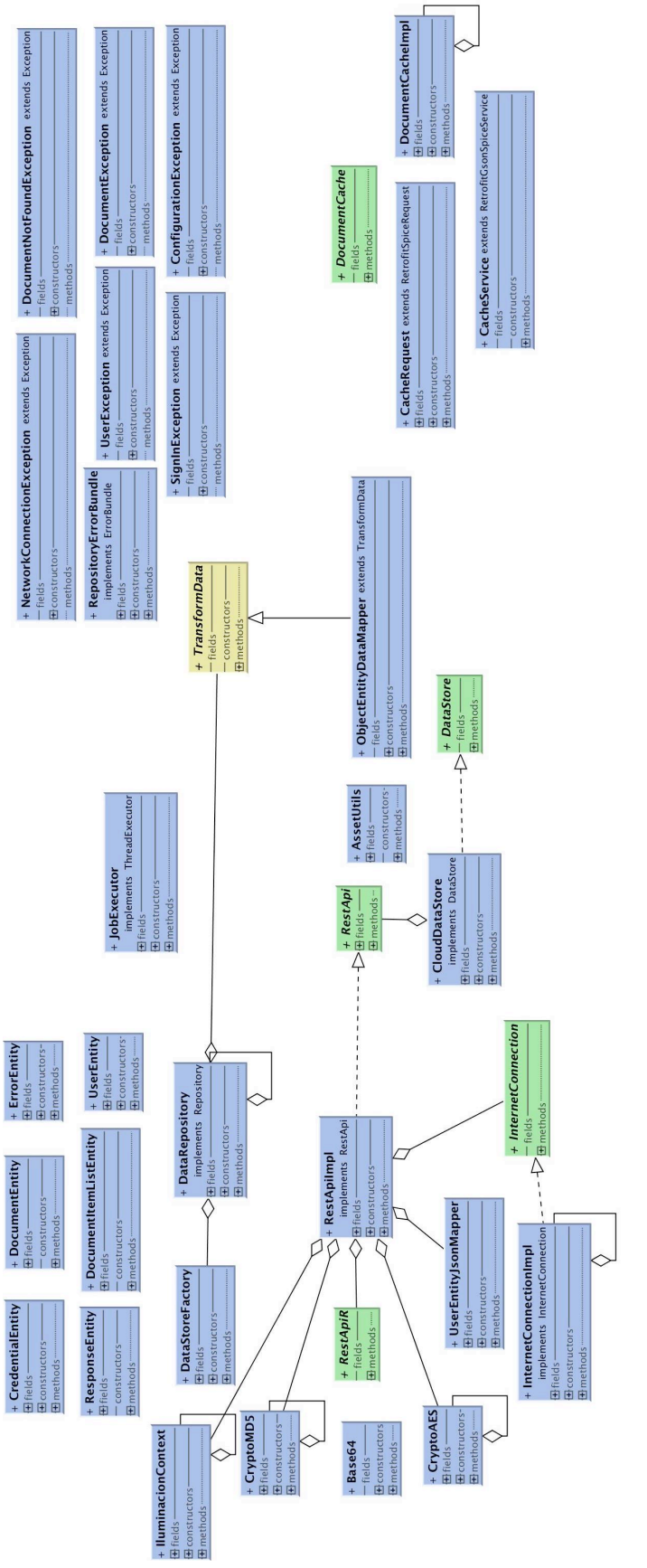

Figura 35. Diagrama de Clases Capa Data.

5.1.3.1 Descripción Estructural

A nivel de dependencias la capa Data corresponde a una librería propia de Android (*apklib*) y está compuesta por los siguientes paquetes:

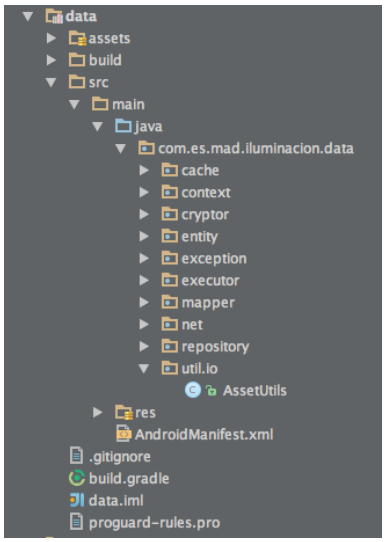

Figura 36. Estructura Data.

- **Paquete Cache**: Es el encargado de gestionar la caché de las peticiones de Partes de Localización Técnica consultados desde la aplicación Android (*Activity DocumentDetailsActivity*).
- **Paquete Context**: Las clases de este paquete se encargan de proveer del contexto de la aplicación, guardando los cambios producidos por el usuario, como por ejemplo guardando las credencias cifradas. Además junto al paquete Util.io carga el contexto inicial del archivo *iluminacion.properties* localizado en el directorio *assets*. En esta primera versión se configura la dirección donde está ubicada la API Rest, el límite máximo de usuarios y partes de localización técnica por cada petición y el token de Google para poder establecer los servicios de notificaciones y consulta de predicción de localizaciones.
- **Paquete Cryptor**: Este paquete se encarga de cifrar y descifrar los tokens necesarios para ser validados en el servidor. Cuando se realizan peticiones al servidor se envía cierta información cifrada y desde el servidor se descifra y valida dicha información para realizar la petición requerida por el usuario a través de la aplicación.
- **Paquete Entity**: Corresponde al modelo de datos de la capa data. Estos objetos que se obtienen o se quieren enviar en formato JSON al servidor son tediosos de controlar de forma manual. Por eso se utiliza *anotaciones* (*GSON* [23]) en las clases de estas entidades para obtener los argumentos a partir de un JSON sin necesidad de recorrer el propio JSON (deserialización) o formar un JSON a partir de un objeto (serialización).
- **Paquete Exception**: Clases que extienden de la clase java *Exception* y son creadas cuando se produce algún error al obtener los datos y transmitidas hacia la capa de dominio.
- **Paquete Executor**: En el encontramos uno de las clases cruciales del proyecto, llamada *JobExecutor* basado en la propuesta del patrón *Singleton* de *Bill Pugh* (*Initialization on Demand Holder*). Esta propuesta surgió antes de la versión 5 de Java cuyo modelo de memoria [24] daba muchos problemas y que afectaban en ciertos escenarios a la hora de obtener instancias de alguna clase *Singleton* de forma simultánea. *Bill Pugh* propuso crear dentro de la clase *Singleton* una clase auxiliar estática interna privada que contiene la instancia de la clase *Singleton* y esta sólo se carga en memoria cuando desde otra parte del código llama a *getInstance()*. Se detallará el uso de esta clase en la sección 5.1.3.2.
- **Paquete Mapper**: Como se explicó en secciones anteriores una de las pautas a seguir para una arquitectura limpia es que cada capa tenga su modelo de datos para

no cruzar el modelo de datos. Y para transmitir el modelo de datos de la capa de datos a la capa de dominio utilizamos las clases de este paquete.

- **Paquete Net**: Las clases que están en dicho paquete son las encargadas de gestionar las peticiones al servidor y controlar y notificar de todo error que se produzca a la hora de obtener los datos.
- **Paquete Repository**: Como se mencionó en la sección 5.1.2.2 implementa el patrón *Repository* mediante la interfaz de la capa de dominio. A través de la estrategia de una factoría provee de los datos a la capa de dominio.

5.1.3.2 JobExecutor

CAPA DOMAIN

```java
public interface ThreadExecutor {

    /**
     * Executes a {@link Runnable}.
     *
     * @param runnable The class that implements {@link Runnable} interface.
     */
    void execute(final Runnable runnable);
}
```

CAPA DATA

```java
public class JobExecutor implements ThreadExecutor {

    private static class LazyHolder {
        private static final JobExecutor INSTANCE = new JobExecutor();
    }

    public static JobExecutor getInstance() { return LazyHolder.INSTANCE; }

    private static final int INITIAL_POOL_SIZE = 3;
    private static final int MAX_POOL_SIZE = 5;

    // Sets the amount of time an idle thread waits before terminating
    private static final int KEEP_ALIVE_TIME = 10;

    // Sets the Time Unit to seconds
    private static final TimeUnit KEEP_ALIVE_TIME_UNIT = TimeUnit.SECONDS;

    private final BlockingQueue<Runnable> workQueue;

    private final ThreadPoolExecutor threadPoolExecutor;

    private JobExecutor() {
        this.workQueue = new LinkedBlockingQueue<Runnable>();
        this.threadPoolExecutor = new ThreadPoolExecutor(INITIAL_POOL_SIZE, MAX_POOL_SIZE,
                KEEP_ALIVE_TIME, KEEP_ALIVE_TIME_UNIT, this.workQueue);
    }

    /**
     * {@inheritDoc}
     *
     * @param runnable The class that implements {@link Runnable} interface.
     */
    @Override public void execute(Runnable runnable) {
        if (runnable == null) {
            throw new IllegalArgumentException("Runnable to execute cannot be null");
        }
        this.threadPoolExecutor.execute(runnable);
    }
}
```

Figura 29. Clase JobExecutor y ThreadExecutor.

Como hemos mencionado anteriormente la clase *JobExecutor está* implementada con un patrón *Singleton* bajo demanda. Esta clase gracias a *ThreadPoolExecutor* permite configurar y agrupa tareas ejecutables (*Runnables*) en una cola de trabajo. Esto permite realizar el trabajo de forma asíncrona, que es un aspecto importante que se tiene que aprovechar en las máquinas multi-core y en la computación en la nube. *ThreadPool-Executor* proporciona varios parámetros ajustables como son (véase Figura 37):

- **INITIAL_POOL_SIZE**: Número inicial de hilos que permanecen fijos. Si llega una tarea nueva (*execute(Runnable runnable)*) y hay menos hilos que INITIAL_POOL_SIZE se crea un nuevo hilo, incluso si hay subprocesos inactivos.
- MAX_POOL_SIZE: Número máximo de hilos que pueden permanecer en el contenedor. Solo se creará otro hilo si no supera MAX_POOL_SIZE y la cola no está llena.
- **KEEP_ALIVE_TIME**: Si el contenedor de hilos supera INITIAL_POOL_SIZE de hilos, los sobrantes serán finalizados si superan el KEEP_ALIVE_TIME teniendo en cuenta la unidad de tiempo KEEP_ALIVE_TIME_UNIT.
- **BlockingQueue**: Evita errores a la hora de sacar datos de una cola que está vacía o al introducir datos cuando ésta se encuentra llena. Esta cola interactúa con el contenedor de hilos de tal modo que si hay menos hilos que INITIAL_POOL_SIZE ejecutándose, *ThreadPoolExecutor* solicita la adicción de un nuevo hilo que añadir a la cola de tareas. Otra de las interacciones entre la *BlockingQueue* y *ThreadPoolExecutor* es, si el número de hilos que se están ejecutando son iguales o superan el INITIAL_POOL_SIZE, *ThreadPoolExecutor* solicita añadir la tarea a la cola que añadir un nuevo hilo. Y por último si la tarea no puede almacenarse en la cola se creará un nuevo hilo mientras que no se exceda el MAX_POOL_SIZE sino será rechazada la tarea.

- **LinkedBloquingQueue**: Es una de las tres estrategias que implementan la interfaz BlockingQueue. Este tipo de colas generan tareas en la cola cuando el número de INITIAL_POOL_SIZE están ocupados, por lo que no MAX_POOL_SIZE no tendrá ningún efecto. Se ha optado por este tipo de cola ya que ninguna tarea depende de otra y no afectan a la ejecución del resto.

5.1.3.3 Librerías

- ***Retrofit***

Retrofit [21] es un cliente http para Android y Java desarrollado por Square, Inc. El código está disponible en el repositorio GitHub [25]. Para la solución IluminaciónFM App se ha declarado una interfaz (*RestApiR*) con las anotaciones pertinentes para realizar las peticiones al servidor (véase Figura 38)

```
public interface RestApiR {


    @POST("/users/login")
    UserEntity signIn(@Header("password") String password, @Body JSONObject credential);


    @POST("/users/create")
    ResponseEntity createUser(@Header("email") String email, @Header("password") String password, @Body JSONObject user);

    @DELETE("/users/delete")
    ResponseEntity deleteUser(@Header("email") String email, @Header("password") String password,
                              @Header("userId") String userId);

    @GET("/users/users")
    List<UserEntity> getUsers(@Header("email") String email, @Header("password") String password,
                              @Header("before") String beforeId, @Header("limit") int limitUsers);
```

Figura 38. Interfaz RestApiR.

Se ha implementado Retrofit de forma síncrona ya que las peticiones se realizan en un contexto fuera del proceso de la interfaz del usuario gracias a *JobExecutor*. Si se deseara realizar una estrategia asíncrona Retrofit da soporte añadiendo como argumento un *Callback*. La implementación de esta interfaz se instancia en el constructor de la clase *RestApiImpl* y utilizada de forma transparente para el desarrollador (véase Figura 39).

Figura 39. de Retrofit en IluminaciónFM APP.

- ***RoboSpice***

RoboSpice es una biblioteca que además de ser un cliente http, tiene almacenamiento en cache de peticiones y junto a Retrofit hacen que la aplicación mejore en rendimiento y reduzca las peticiones. Es un proyecto open source iniciado por Octo Technology y su código fuente se puede consultar en el repositorio GitHub [22]. Para esta primera versión se ha utilizado para cachear peticiones de mostrar un Parte de Localización Técnica.

Primero se configura el servicio de robospice detallando aspectos como url del servidor (véase Figura 40) si el Parte de localización Técnica no se encuentra en caché solicitar a través de retrofit, pero esa parte es transparente al programador como podemos ver en la Figura 41.

Figura 40. Clase CacheService.

Figura 41. Cache de Peticiones CacheRequest.

La clase que gestiona la cache es *DocumentCacheImpl* implementada a través de un patrón Singleton y desde la clase *RestApiImpl* forman la petición del Parte de Localización Técnica (véase Figura 42).

Figura 42. Implementación *getDocumentDetails* en *RestApiImpl*.

5.1.5 Comportamiento (Diagrama de Secuencia)

Para completar el estudio de la implementación de una arquitectura limpia a nivel de software se ha puesto como ejemplo este diagrama de secuencia (véase Figura 43) a la hora de crear un usuario

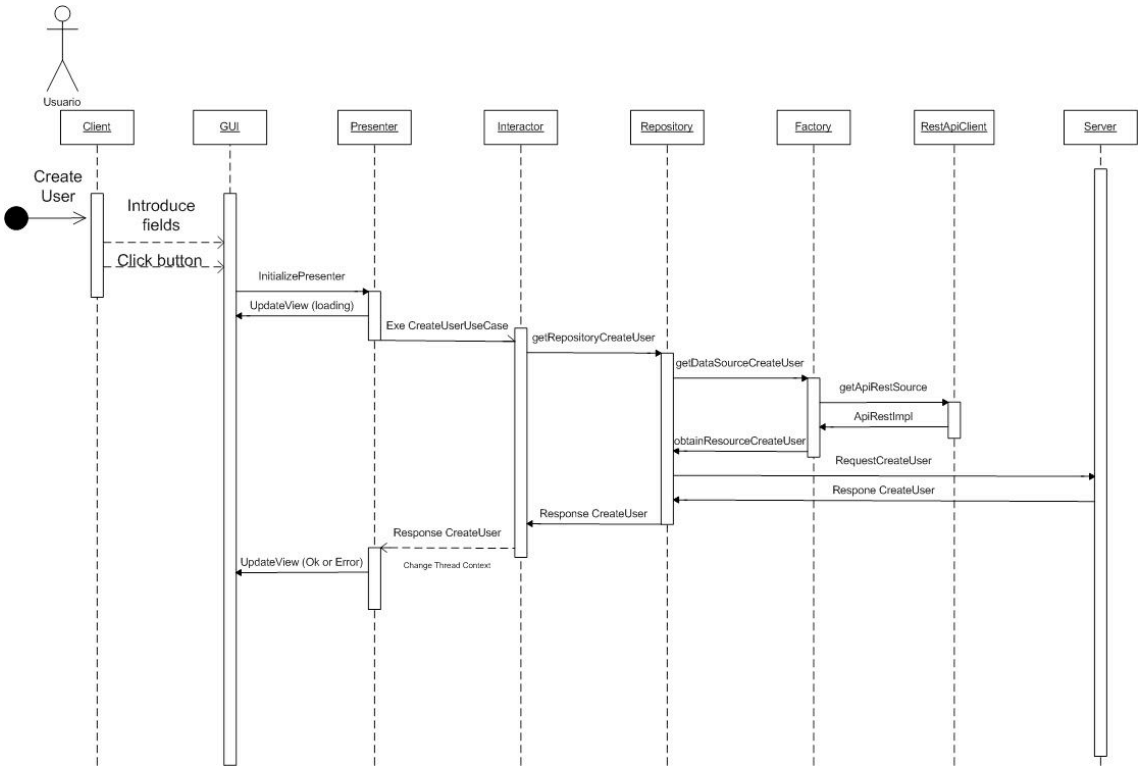

Figura 43. Diagrama de secuencia creación de un usuario.

5.2 API Rest

En esta primera versión de IluminaciónFM se ha implementado los siguientes servicios para la gestión de usuarios y de Partes de Localización Técnica:

5.2.1 API Rest Users.

Method	Uri Pattern	Controller#Action	Description
Post	/users/login	api/users#login	Registra un dispositivo a un usuario
Post	/users/create	api/users#create	Crea un nuevo usuario.
Post	/users/config_pass_user	api/users#config_pass_user	Cambia la contraseña a un usuario.
Delete	/users/delete	api/users#delete	Elimina a un usuario del sistema.
Get	/users/retrieve_pass_user	api/users#retrieve_pass_user	Genera nueva contraseña a un usuario y se le envía a su correo.
Get	/users/users	api/users#users	Devuelve un número de

			usuarios dependiendo del campo *limit* y a partir de un usuario gracias al campo *before*.
Get	/users/logout	api/users#logout	Elimina asociación entre usuario y dispositivo.

Tabla 1. Información detallada API Rest Users.

5.2.2 API Rest Technical Document Location.

Method	Uri Pattern	Controller#Action	Description
Post	/technical_document_locations/create	api/technical_document_locations#create	Crea un Parte de Localización Técnica
Get	/technical_document_locations/documents	api/technical_document_locations#documents	Devuelve un número de partes dependiendo del campo *limit* y a partir de un parte gracias al campo *before*.
Get	/technical_document_locations/document	api/technical_document_locations#document	Devuelve un Parte de Localización Técnica.
Delete	/technical_document_locations/delete	api/technical_document_locations#delete	Elimina un Parte de Localización Técnica.

Tabla 2. Información detallada API Rest Technical Document Location.

5.3 Renderizado PDF

Gracias a la librería *Wicked PDF* ha permitido generar documentos pdf a partir de una plantilla en html. Esta librería creada por *Miles Z. Sterrett* la podemos encontrar en el repositorio GitHub [6].

El uso de esta librería es sencillo, simplemente he creado una vista en html con el mismo aspecto que el Parte de Localización Técnica del **Anexo I.** Como ya sabemos Ruby on Rails maneja el paradigma del MVC (Model-View-Controller) por lo que desde el controlador se declara una variable de instancia (véase Figura 44) que permite a la acceder a los atributos del Parte de Localización Técnica.

Controlador

```
def createPDF(document)
    @document_pdf = document
    pdf = render_to_string pdf: "renderPDF", template: "/layouts/technical_document_location.html.erb"

    file_pdf = File.open("Documento_#{document._id}.pdf", 'wb') do |file|
      file << pdf
    end
end
```

Vista

```
<div id="left">
  <a class="thick">PRODUCTORA: </a><a><%= @document_pdf.film_producer %></a></p>
  <a class="thick">PRODUCIÓN: </a><a><%= @document_pdf.production %></a></p>
  <a class="thick">FECHA DE RODAJE: </a><a><%= @document_pdf.filming_date.strftime("%d-%m-%Y") %></a></p>
  <a class="thick">TOTAL DIAS RODAJE: </a><a><%= @document_pdf.total_filming_days %></a></p>
  <a class="thick">DIRECTOR DE FOTOGRAFIA: </a><a><%= @document_pdf.director_photography %></a></p>
```

Figura 44. Utilización de la librería Wicke PDF.

Tras el renderizado del Parte de Localización Técnica se guarda en el servidor y se envía a los correos de los administradores y una vez finalizados los envíos se elimina el pdf renderizado.

5.4 Colas de Mensaje

Uno de los problemas que surgieron relacionado con el servidor era el tiempo de respuesta que se daba al dispositivo a la hora de crear un Parte de Localización Técnica ya que se creaba un pdf a partir de una plantilla en html y el envío de dicho pdf a través del correo a los diferentes administradores. Para quitar carga de procesamiento a la API de iluminaciónFM se optó por la utilización de colas de mensaje. En esta primera versión se ha introducido el uso de colas de mensaje dentro del proyecto de iluminaciónFM y queda pendiente como trabajo futuro extraerlo a un nuevo proyecto permitiendo facilitar la escalabilidad del proyecto.

En el controlador *TechnicalDocumentLocationsController* una vez que se introduce los campos que conforman el Parte de Localización Técnica en la base de datos se envía

Figura 45. Creación y envío de tarea para la cola de mensajes RabbitMQ.

una tarea a la cola de mensajes (véase Figura 45). La tarea envía un hash con las claves *technical_document_location_id*, *user_name*, *user_second_name* con sus respectivos valores y serán recogidos por el suscriptor quien realizará las operaciones con mayor carga computacional liberando el proceso de respuesta de peticiones de la API Rest. Como podemos ver en la Figura 45 en el método *sendToRabbit* cuando se publica la tarea (*q.publish(hash, :persistent => true*) le decimos que sea persistente en el sistema, por si en el momento de enviarse o cuando la procesa el suscriptor hay un error, esta tarea no se perderá y cuando se vuelva a enviar a la cola de mensajes una nueva tarea volverá a intentar todas aquellas tareas que no se pudieron realizar con anterioridad.En la estructura de un proyecto en *Ruby on Rails* podemos encontrar puntos donde colocar código de inicialización. En el directorio **config/** encontramos el script *application.rb* que permite dicha inicialización y ha permitido ubicar el suscriptor de la cola de mensajes, es decir, el que va a realizar el renderizado del pdf y su envio a los diferentes emails y además enviará las notificaciones push a los dispositivos móviles (véase Figura 46).

Script config/application.rb

```ruby
Thread.new do
    conn  = Bunny.new(:hostname => "localhost")
    conn.start
    ch    = conn.create_channel
    q     = ch.queue("renderSendPDF")
    q.subscribe(:ack => true, :block => true) do |delivery_info, properties, body|
      puts " [x] Received #{body}"
      my_hash = JSON.parse("#{body}")

      document_id = my_hash["technical_document_location_id"].to_s
      user_name = my_hash["user_name"].to_s
      user_second_name = my_hash["user_second_name"].to_s
      # imitate some work
      # sleep body.count(".").to_i
      # puts " [x] Done"
      documents = TechnicalDocumentLocation.where(:_id => document_id)

      if documents
          document = documents.first
          foo = ActionController::Base::ApplicationController.new
          foo.createPDF(document)
          admins = User.where(:role => "admin")
          foo.saveDocumnetNotRead(admins)
          foo.sendEmail(admins, document)
          foo.sendNotificationsPush(user_name, user_second_name, document._id)
          foo.deletePDF(document_id)
      end

      ch.ack(delivery_info.delivery_tag)
    end
    conn.close
end
```

Figura 46. Suscriptor de la cola de mensajes que realiza las operaciones de mayor carga computacional.

RabbitMQ permite monitorizar las colas de mensajes gracias a una interfaz gráfica que ofrece RabbitMQ. Para acceder a ella basta con introducir en nuestro navegador la url de nuestro servidor que tenga instalado RabbitMQ y añadir el puerto 15672 que tiene por defecto. Tras introducir las credenciales que pide RabbitMQ tendremos a nuestra disposición detalles acerca del estado de nuestro sistema de colas (véase Figura 47).

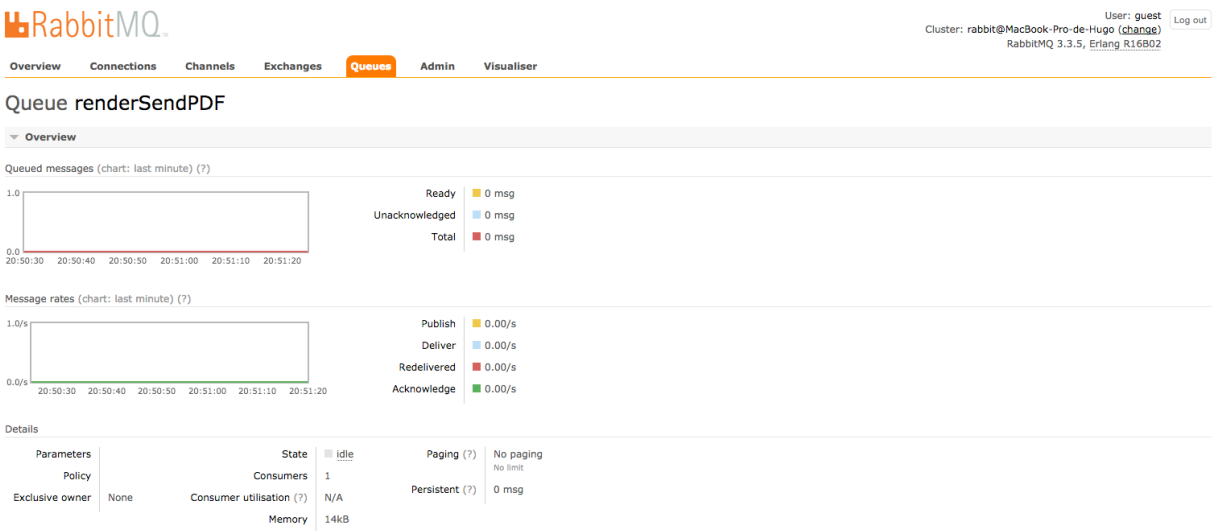

Figura 47. Estado de la cola de mensajes renderSendPDF.

6. Pruebas

Antes de salir a producción, una aplicación debe pasar por una batería de test para probar el producto que se va a entregar al cliente. El objetivo de las pruebas es confirmar que el producto que se entrega es fiable.

Para IluminaciónFM App para Android se han realizado pruebas unitarias e integración de las diferentes capas del proyecto. Como se puede observar el dividir el proyecto en capas simplifica las pruebas y permite agilizar el desarrollo en un equipo de programadores que trabajen en funcionalidades para diferentes módulos. Evitando estar a expensas del avance de otras capas. Además ayuda a definir conceptualmente el cometido de cada una de las capas y no romper la regla de dependencia evitando por ejemplo tener que verificar comportamientos de la interfaz al realizar un test de la capa de datos.

Bloque	Paquete	Nº de Tests
APP	Presenter	11
DOMAIN	Interactor	55
DATA	Exception	1
	Mapper	3
	Repository	57
	TOTAL	127

Tabla 3. Test Unitarios de IluminaciónFm App Android.

Para la realización de los diferentes test se han utilizado las siguientes librerías:

- **Mockito**: Mockito es una librería open source [26] y sirve para realizar test unitarios, es decir, para probar las responsabilidades de cada clase. Pero estas clases a su vez están relacionadas con otras clases que tienen que ser instanciadas para que todo funcione correctamente. Por lo que Mockito permite crear objetos que imitan el comportamiento (dobles) de aquellos objetos que colaboran con la clase que se quiere testear, programando el comportamiento de los colaboradores y verificando las interacciones. Gracias a Mockito podemos independizar el desarrollo de la aplicación ya que podemos definir mocks de objetos que todavía no están disponibles. Así se puede desarrollar una parte de un bloque de la aplicación y verificar que funciona correctamente sin dependencia del resto de otras partes.
- **JUnit**: Uno de los frameworks open source más popular para realizar tests en Java [27]. Se ha utilizado la versión JUnit 4 que a diferencia de las versiones anteriores sí incluye anotaciones para agilizar el desarrollo de tests. Las más importantes son:

- ***@RunWith:*** Se le asigna a una clase a la que JUnit invocará para que ejecute los tests en lugar de llamar al ejecutor por defecto de JUnit.
- ***@Before:*** Indica que el siguiente método se debe ejecutar antes de cada test.
- ***@After:*** Indica que el siguiente método debe ser ejecutado después de cada test.
- ***@Test:*** Indica a JUnit que dicho método debe ser testeado. En versiones anteriores el método debía de comenzar por la palabra "*test*" pero gracias a esta anotación se puede elegir cualquier nombre para nuestro test.

6.1 Capa Presentación

Para realizar estas pruebas unitarias se ha utilizado la librería *Mockito* y se van a realizar pruebas sobre las actividades, presentadores y transformadores de datos. En esta capa se han realizado test unitarios para la inicialización de los presentadores verificando la correcta ejecución de los respectivos casos de uso (véase Figura 48).

```
public class ConfigUserPresenterTest extends InstrumentationTestCase {

    private static final String TEST_USER_PASSWORD =    "12345";

    private ConfigUserPresenter configUserPresenter;

    @Mock
    private ConfigPassUserView mockConfigPassUserView;
    @Mock
    private ChangeConfigurationUserUseCase mockChangeConfigurationUseCase;


    @Override protected void setUp() throws Exception {

        super.setUp();
        //Context context = getInstrumentation().getContext();
        MockitoAnnotations.initMocks(this);
        configUserPresenter = new ConfigUserPresenter(mockConfigPassUserView, mockChangeConfigurationUseCase);
    }

    public void testConfigUserPresenterTestInitialize() {
        doNothing().when(mockChangeConfigurationUseCase)
                .execute(anyString(), any(ChangeConfigurationUserUseCase.Callback.class));
        given(mockConfigPassUserView.getContext()).willReturn(getInstrumentation().getContext());

        configUserPresenter.initialize(TEST_USER_PASSWORD);

        verify(mockConfigPassUserView).hideRetry();
        verify(mockConfigPassUserView).showLoading();
        verify(mockChangeConfigurationUseCase).execute(anyString(), any(ChangeConfigurationUserUseCase.Callback.class));
    }
```

Figura 48. Test Unitario Prsentador *ChangeConfigurationUserUSeCase.*

En el test de ejemplo mostrado en la Figura 48 se prueba la inicialización del presentador (*configureUserPresenter.initialize(TEST_USER_PASSWORD)*) y verifica que los *Mocks* de la vista (*mockConfigPassUserVIew*) y caso de uso (*ChangeConfiguration-UserUseCase*) han realizado dicho comportamiento (*hideRetry, showloading*, etc).

6.2 Capa Dominio

Para realizar estas pruebas unitarias se ha utilizado Mockito y JUnit. En este bloque se han realizado test sobre los casos de uso que se han implementado. A continuación se muestra la realización de test unitarios para el caso de uso *GetDocumentListUseCase:*

```
public class GetDocumentListUseCaseTest {

    private static String EMPTY_BEFORE = "";
```

```
    private static int ZERO_USERS = 0;
    private static int NUM_USERS = 10;
    private GetDocumentListUseCase getDocumentListUseCase;

    @Mock
    private ThreadExecutor mockThreadExecutor;
    @Mock
    private PostExecutionThread mockPostExecutionThread;
    @Mock
    private Repository mockUserRepository;


    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
        getDocumentListUseCase = new GetDocumentListUseCaseImpl(mockUserRepository,
mockThreadExecutor,
                mockPostExecutionThread);
    }

    @Test
    public void testGetDocumentListUseCaseExecution() {
        doNothing().when(mockThreadExecutor).execute(any(Interactor.class));

        GetDocumentListUseCase.Callback mockGetDocumentListCallback =
mock(GetDocumentListUseCase.Callback.class);

        getDocumentListUseCase.execute(EMPTY_BEFORE, NUM_USERS,
mockGetDocumentListCallback);

        verify(mockThreadExecutor).execute(any(Interactor.class));
        verifyNoMoreInteractions(mockThreadExecutor);
        verifyZeroInteractions(mockUserRepository);
        verifyZeroInteractions(mockPostExecutionThread);
    }

    @Test
    public void testGetDocumentListUseCaseInteractorRun() {
        GetDocumentListUseCase.Callback mockGetDocumentListCallback =
mock(GetDocumentListUseCase.Callback.class);

        doNothing().when(mockThreadExecutor).execute(any(Interactor.class));
        doNothing().when(mockUserRepository).getDocumentList(anyString(), anyInt(),
any(Repository.DocumentListCallback.class));

        getDocumentListUseCase.execute(EMPTY_BEFORE, NUM_USERS,
mockGetDocumentListCallback);
        getDocumentListUseCase.run();

        verify(mockUserRepository).getDocumentList(anyString(), anyInt(),
any(Repository.DocumentListCallback.class));
        verify(mockThreadExecutor).execute(any(Interactor.class));
        verifyNoMoreInteractions(mockUserRepository);
        verifyNoMoreInteractions(mockThreadExecutor);
    }

    @Test
    @SuppressWarnings("unchecked")
    public void testUserListUseCaseCallbackSuccessful() {
        final GetDocumentListUseCase.Callback mockGetDocumentListCallback =
```

```
            mock(GetDocumentListUseCase.Callback.class);
        final Collection<Document> mockResponseDocumentList = (Collection<Document>)
mock(Collection.class);

        doNothing().when(mockThreadExecutor).execute(any(Interactor.class));
        doAnswer(new Answer() {
            @Override public Object answer(InvocationOnMock invocation) throws Throwable {
                ((Repository.DocumentListCallback) invocation.getArguments()[2]).onDocumentListLoaded(
                        mockResponseDocumentList);
                return null;
            }
        }).when(mockUserRepository).getDocumentList(anyString(), anyInt(),
any(Repository.DocumentListCallback.class));

        getDocumentListUseCase.execute(EMPTY_BEFORE, NUM_USERS,
mockGetDocumentListCallback);
        getDocumentListUseCase.run();

        verify(mockPostExecutionThread).post(any(Runnable.class));
        verifyNoMoreInteractions(mockGetDocumentListCallback);
        verifyZeroInteractions(mockResponseDocumentList);
    }

    @Test
    public void testUserListUseCaseCallbackError() {
        final GetDocumentListUseCase.Callback mockGetDocumentListUseCaseCallback =
                mock(GetDocumentListUseCase.Callback.class);
        final ErrorBundle mockErrorBundle = mock(ErrorBundle.class);

        doNothing().when(mockThreadExecutor).execute(any(Interactor.class));
        doAnswer(new Answer() {
            @Override public Object answer(InvocationOnMock invocation) throws Throwable {
                ((Repository.DocumentListCallback)
invocation.getArguments()[2]).onError(mockErrorBundle);
                return null;
            }
        }).when(mockUserRepository).getDocumentList(anyString(), anyInt(),
any(Repository.DocumentListCallback.class));

        getDocumentListUseCase.execute(EMPTY_BEFORE, NUM_USERS,
mockGetDocumentListUseCaseCallback);
        getDocumentListUseCase.run();

        verify(mockPostExecutionThread).post(any(Runnable.class));
        verifyNoMoreInteractions(mockGetDocumentListUseCaseCallback);
        verifyZeroInteractions(mockErrorBundle);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testExecuteDocumentrListUseCaseNullParameter() {
        getDocumentListUseCase.execute(null, 0, null);
    }
}
```

Código 1. Clase *GetDocumentListUseCaseTest.*

Como podemos ver en el Código 1 se utilizan las anotaciones de Mockito (@Mock) y de JUnit 4 (@Test, @Before) y se realizan 5 tests tras previamente instanciar la clase a probar (*setup()*) junto a los mocks (*MockitoAnnotations.initMock(this)*):

- ***testGetDocumentListUseCaseExecution***: La primera sentencia determina que el mock *ThreadExecutor* no continúe con su ejecución cuando invoque su método *execute*. La segunda sentencia crea un mock con el callback que devuelve la lista de Partes de Localización Técnica y que utilizará el caso de uso en la tercera sentencia. Una vez que se realiza la tercera sentencia de este test se verifica que el mock *mockThreadExecutor* invoca su método *execute* en el método *execute* de la clase *GetDocumentListCaseImpl*. También comprueba que solo hay una única invocación del mock *mockThreadExecutor* y ninguna por parte de los mocks *mockUserRepository y mockPostExecutionThread* en el método *execute* de la clase *GetDocumentListCaseImpl.*
- ***testGetDocumentListUseCaseInteractorRun****:* Se comprueba las interacciones en el método *run()* de la clase *GetDocumentListCaseImpl.* Para ello no se permite que siga el flujo de la ejecución cuando se invoca el método *execute()* de *mockThreadExecutor* y *getDocumentList(...)* de la clase que implemente la interfaz *Repository (mockUserRepository).* Además comprueba que solo hay una única invocación de los métodos de las clases anteriormente citadas.
- ***testGetDocumentListUseCaseCallbackSuccesfull****:* En este test se simula un caso de éxito de la clase *GetDocumentListCaseImpl.* Para ello se crea una respuesta de éxito (una colección de *Document*) por parte de uno de los callbacks de la interfaz *Repository (doNothing(...))* cuando solicita Partes de Localización Técnica al repositorio. Y una vez establecida la situación se verifica que únicamente hay una sola invocación al callback que obtiene la lista de Partes por parte del repositorio y que se envían a través del callback del caso de uso *GetDocumentListCase (Callback.onDocumentListLoaded())* al hilo de la interfaz de usuario.
- ***testGetDocumentListUseCaseCallbackError****:* Este test simula un caso de error de la clase *GetDocumentListCaseImpl.* Para ello se crea una respuesta de error por parte de uno de los callbacks de la interfaz *Repository (doNothing(...))* cuando solicita Partes de Localización Técnica al repositorio. Y una vez establecida la situación se verifica que únicamente hay una sola invocación al callback que obtiene una respuesta de error por parte del repositorio y que se envían a través del callback del caso de uso *GetDocumentListCase (Callback.onError())* al hilo de la interfaz de usuario.
- ***testExecuteDocumentListUseCaseNullParameter****:* Comprueba gracias a la anotación de JUnit 4 (*@Test(expected = IllegalArgumentException.class*)) que al ejecutar un caso de uso con campos a null lanza una excepción *IllegalArgumentException.*

6.3 Capa Datos

Para realizar estas pruebas se ha utilizado Mockito y JUnit 4. Los test principales que se han realizado han sido los relacionados con las clases *CloudDataStore* y *DataRepository.*

6.3.1 CloudDataStore

```
public class CloudDataStoreTest extends ApplicationTestCase{
   private static final String TEST_BEFORE =          "123456";
   private static final int TEST_LIMIT =              20;
```

```
    private CloudDataStore cloudDataStore;

    @Mock
    private RestApi mockRestApi;
    @Mock
    private DataStore.DocumentListCallback mockDocumentListDataStoreCallback;

    @Captor
    private
ArgumentCaptor<RestApi.DocumentListCallback>restApiDocumentListCallbackArgumentCaptor;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
        cloudDataStore = new CloudDataStore(mockRestApi);
    }

@Test
public void testGetDocumentListSuccessfully() {
    List<DocumentItemListEntity> mockDocumentEntityCollection =
(List<DocumentItemListEntity>)mock(List.class);

    cloudDataStore.getDocumentsEntityList(TEST_BEFORE, TEST_LIMIT,
mockDocumentListDataStoreCallback);

    verify(mockRestApi).getDocuments(anyString(), anyInt(),
restApiDocumentListCallbackArgumentCaptor.capture());
    verifyZeroInteractions(mockDocumentListDataStoreCallback);


restApiDocumentListCallbackArgumentCaptor.getValue().onDocumentListLoaded(mockDocumentE
ntityCollection);


verify(mockDocumentListDataStoreCallback).onDocumentListLoaded(mockDocumentEntityCollectio
n);
}
@Test
public void testGetDocumentListError() {
    cloudDataStore.getDocumentsEntityList(TEST_BEFORE, TEST_LIMIT,
mockDocumentListDataStoreCallback);
    verify(mockRestApi).getDocuments(anyString(), anyInt(),
restApiDocumentListCallbackArgumentCaptor.capture());
    verifyZeroInteractions(mockDocumentListDataStoreCallback);
    restApiDocumentListCallbackArgumentCaptor.getValue().onError(any(Exception.class));
    verify(mockDocumentListDataStoreCallback).onError(any(Exception.class));
}
```

Código 2. Test parcial de la clase CloudDataStoreTest.

Siguiendo con los test relacionados con la lista de los Partes de Localización Técnica (véase Código 2) pero esta vez desde la capa de datos. Como podemos apreciar aparece una nueva anotación de Mockito (*@Captor*) cuyo objetivo es obtener un argumento que en nuestro ejemplo sería el callback *DocumentListCallback.onDocumentListLoad (interfaz)* incluido en la interfaz *RestApi.* Tras obtener dicho argumento se verifica que se ha obtenido la lista de Partes de Localización Técnica. En el otro test tiene el mismo objetivo pero capturando como argumento el callback *DocumentListCallback.onError* de la interfaz *RestApi.*

6.3.2 DataRepository

Como ya vimos en secciones anteriores *DataRepository* es quien implementa la interfaz *Repository*. Esta clase es la encargada de transformar los datos de su modelo al modelo de datos de la capa de dominio y la transferencia de los mismos. Contemplando el caso de uso de obtención de una lista de Partes de Localización Técnica se han realizado tres tests como podemos ver en el Código 3:

```
public class DataRepositoryTest extends ApplicationTestCase{

   private static final String TEST_BEFORE =        "123456";
   private static final int TEST_LIMIT =            20;
   private DataRepository dataRepository;
   @Mock
   private DataStoreFactory mockDataStoreFactory;
   @Mock
   private ObjectEntityDataMapper mockObjectEntityDataMapper;
   @Mock
   private DataStore mockDataStore;
   @Mock
   private Document mockDocument;
   @Mock
   private List<Document> mockDocumentList;
   @Mock
   private Response mockResponse;
   @Mock
   private DocumentEntity mockDocumentEntity;
   @Mock
   private List<DocumentItemListEntity> mockDocumentItemListEntity;
   @Mock
   private Repository.DocumentListCallback mockDocumentListRepositoryCallback;
   @Rule
   public ExpectedException expectedException = ExpectedException.none();
   @Before
   public void setUp() {
      MockitoAnnotations.initMocks(this);
      resetSingleton(DataRepository.class);
      dataRepository = DataRepository.getInstance(mockDataStoreFactory,
            mockObjectEntityDataMapper);

      given(mockDataStoreFactory.create()).willReturn(mockDataStore);
      given(mockDataStoreFactory.createCloudDataStore()).willReturn(mockDataStore);
   }

   @Test
   public void testGetDocumentListSuccess(){
      doAnswer(new Answer() {
         @Override
         public Object answer(InvocationOnMock invocation) throws Throwable {
            ((DataStore.DocumentListCallback)
invocation.getArguments()[2]).onDocumentListLoaded(mockDocumentItemListEntity);
            return null;
         }
      }).when(mockDataStore).getDocumentsEntityList(anyString(), anyInt(),
any(DataStore.DocumentListCallback.class));

given(mockObjectEntityDataMapper.transform(mockDocumentItemListEntity)).willReturn(mockD
ocumentList);
```

```
        dataRepository.getDocumentList(TEST_BEFORE, TEST_LIMIT,
mockDocumentListRepositoryCallback);

        verify(mockObjectEntityDataMapper).transform(mockDocumentItemListEntity);
        verify(mockDocumentListRepositoryCallback).onDocumentListLoaded(mockDocumentList);


    }

    @Test
    public void testGetDocumentNullResult(){
        doAnswer(new Answer() {
            @Override
            public Object answer(InvocationOnMock invocation) throws Throwable {
                ((DataStore.DocumentListCallback)
invocation.getArguments()[2]).onDocumentListLoaded(mockDocumentItemListEntity);
                return null;
            }
        }).when(mockDataStore).getDocumentsEntityList(anyString(), anyInt(),
any(DataStore.DocumentListCallback.class));

given(mockObjectEntityDataMapper.transform(mockDocumentItemListEntity)).willReturn(null);


doNothing().when(mockDocumentListRepositoryCallback).onError(any(RepositoryErrorBundle.class)
);

        dataRepository.getDocumentList(TEST_BEFORE, TEST_LIMIT,
mockDocumentListRepositoryCallback);

        verify(mockObjectEntityDataMapper).transform(mockDocumentItemListEntity);
        verify(mockDocumentListRepositoryCallback).onDocumentListLoaded(null);

    }

    @Test
    public void testGetDocumentListByError(){
        doAnswer(new Answer() {
            @Override
            public Object answer(InvocationOnMock invocation) throws Throwable {
                ((DataStore.DocumentListCallback)
invocation.getArguments()[2]).onError(any(Exception.class));
                return null;
            }
        }).when(mockDataStore).getDocumentsEntityList(anyString(), anyInt(),
any(DataStore.DocumentListCallback.class));


        dataRepository.getDocumentList(TEST_BEFORE, TEST_LIMIT,
mockDocumentListRepositoryCallback);

        verify(mockDocumentListRepositoryCallback).onError(any(RepositoryErrorBundle.class));
        verifyZeroInteractions(mockObjectEntityDataMapper);

    }
}
```

Código 3. Test Unitarios Clase *DataStoreRepositoriy* para *getDocumentList.*

- ***testGetDocumentListSuccess***: Se crea una respuesta con una lista de Partes de Localización Técnica cuyo modelo de datos corresponde a la capa de datos. Después se verifica que se transforman los datos gracias a el objeto *mockObjectEntityDataMapper* y que se devuelve a la capa de dominio.
- ***testGetDocumentNull:*** Comprueba lo mismo que el test anterior pero contemplando el caso que se devuelva una colección de Partes vacía.
- ***testGetDocumentListByError***: Se crea una respuesta de error y se verifica que se devuelve dicha respuesta y que no hay ninguna invocación por parte del mock *mockObjectEntityDataMapper*.

6.4 Pruebas de usuario final

Para realizar estas pruebas se han utilizado 5 personas que no habían utilizado previamente la aplicación. Las pruebas se han clasificado en los diferentes contextos y cuyos resultados se encuentran en el **Anexo II**:

- Alta de usuarios.
- Baja de usuarios.
- Recuperación de contraseña.
- Cambio de contraseña.
- Cerrar sesión.
- Crear Parte de Localización Técnica.
- Borrar Parte de Localización Técnica.
- Cargar listado de Partes de Localización Técnica.
- Cargar lista de usuarios.

7. Conclusiones y Trabajo Futuro

Hemos podido ver a lo largo del documento que a la hora de afrontar un proyecto no solo basta con satisfacer los requisitos. Antes debemos tener en cuenta buenas prácticas y pautas de diseño a la hora de estructurar un proyecto de tal modo que nos ayude a agilizar todo proceso de desarrollo. Gracias a las arquitecturas Hexagonal y Onion han permitido que el proyecto IluminaciónFM App para Android sea fácil de mantener, fácil de testear, fácil de añadir nuevos requisitos y que no dependa de ningún framework.

Por otro lado a nivel técnico he aprendido nuevos patrones como MVP (Model-View-Presenter) y *Repository* que me han ayudado a abstraer más un proyecto. También he adquirido nuevos conocimientos relacionados con el entorno de servicios, desarrollando una API en Ruby on Rails y colas de mensaje de RabbitMQ para mejorar el rendimiento. He utilizado nuevos frameworks como Mockito y JUnit 4 para la realización de tests en la aplicación Android dándole al proyecto una mayor calidad. Además he utilizado por primera vez una base de datos NoSQL en concreto MongoDB que permite hacer consultas geoespaciales muy útiles en el futuro para la empresa Iluminación FM.

Como resultado de este proyecto ha permitido digitalizar un proceso muy importante por parte de IluminaciónFM a la hora de hacer un presupuesto inicial.

Como trabajo futuro se comienza con una nueva fase cuyo objetivo es incluir otro tipo de estudio en el cual estimar personal y transporte según que servicio tengan que proveer a sus clientes.

A nivel técnico realizaré un estudio sobre una nueva alternativa relacionado con ejecuciones asíncronas y es la utilización de RxJava ya disponible en la versión 1.8 Java. Aplicaciones como SoundCloud ya utilizan dicha alternativa.

Otro objetivo es abstraer un nivel más esta aplicación para otras empresas que quieran digitalizar un proceso propio de la empresa que requiera completar un formulario con papel y bolígrafo.

REFERENCIAS

[1] Alistair Cockburn. (2008, April 1). Hexagonal architecture., Sitio web: http://alistair.cockburn.us/Hexagonal+architecture

[2] Jeffrey Palermo. (2008, July 29). The Onion Architecture., Sitio web: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

[3] Google. (2015, August 26). Google Cloud Messaging. Retrieved February 20, Sitio web https://developers.google.com/cloud-messaging/

[4] Google. (2015, August 26). Google Cloud Messaging. Retrieved February 20, Sitio web https://developers.google.com/cloud-messaging/android/start

[5] Jeff Carbonella, Nathan Herald, Justin Grevich, Alexandre Loureiro Solleiro, Johnny Shields, Jeff Carbonella. (20 Oct 2010). Gmaln for Ruby, de GmailGem team Sitio web: https://github.com/gmailgem/gmail

[6] David Jones. (24 Apr 2013). Wicked PDF, Sitio web: https://github.com/mileszs/wicked_pdf

[7] IDC. (2015). Smartphone OS Market Share, 2015 Q2, de IDC Sitio web: http://www.cva.itesm.mx/biblioteca/pagina_con_formato_version_oct/apaweb.html

[8] Google. Android Studio SDK, de Google Sitio web: http://developer.android.com/intl/es/sdk/index.html

[9] Graeme Rocher, Grails. (2001). IntelliJ IDEA, de JetBrains Sitio web: https://www.jetbrains.com/idea/

[10] Wikipedia. (11 Jun 2015). Lint, de Wikipedia Sitio web: https://es.wikipedia.org/wiki/Lint

[11] Robert Martin (Uncle Bob). (13 Aug 2012). The clean Architecture. , de 8th light Sitio web: http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

[12] Martin Fowler. (18 July 2006). Passive View, de Blog Martin Fowler Sitio web: http://martinfowler.com/eaaDev/PassiveScreen.html

[13] Martin Fowler. Repository, de Blog Matin Fowler Sitio web: http://martinfowler.com/eaaCatalog/repository.html

[14] Square. (29 Jun 2012). Otto, de Square Sitio web: http://square.github.io/otto/

[15] Robert C. Martin. (2012). Código Limpio. España: Anaya.

[16] Aidan Follestad. (16 Jan 2015). Material Dialogs, de Github Sitio web: https://github.com/afollestad/material-dialogs

[17] Sergej Shafarenka. (15 Nov 2013). Material Dialogs, de Github Sitio web: https://github.com/beworker/pinned-section-listview

[18] Shafarenka, S. (2013). *Pinned Section for Android ListView (versión 1.6)* [Video]. Disponible en: https://www.youtube.com/watch?v=mI3DpuoIIhQ

[19] CiTuX. (19 Jun 2014). DateTimePicker, de GitHub Sitio web: https://github.com/CiTuX/datetimepicker

[20] Daimajia. (23 Jun 2014). Android View Animations, de GitHub Sitio web: https://github.com/daimajia/AndroidViewAnimations

[21] Square. (13 May 2013). Retrofit, de Aquare Sitio web: http://square.github.io/retrofit/

[22] Stéphane Nicolas & Octo Technology. (20 Sep 2012). RoboSpice, de GitHub Sitio web: https://github.com/stephanenicolas/robospice

[23] Inderjeet Singh, Joel Leitch, Jesse Wilson. Gson User Guide, de Google Sitio web: https://sites.google.com/site/gson/gson-user-guide

[24] William Pugh. The Java Memory Mode, de Sitio web: http://www.cs.umd.edu/~pugh/java/memoryModel/

[25] Square. (13 May 2013). Retrofit, de GitHub Sitio web: https://github.com/square/retrofit

[26] Google (14 Jan 2009). Mockito de GitHub Sitio Web: https://github.com/mockito/mockito

[27] JUnit. JUnit de Sitio Web: http://junit.org/index.html

ANEXO I

PARTE DE LOCALIZACIÓN TÉCNICA

ILUMINACION

FECHA:

CLIENTE:

PARTE DE LOCALIZACION TÉCNICA

PRODUCCIÓN:		RPBLE. DE PRODUCCIÓN:	
FECHA DE RODAJE:		MAIL CONTACTO:	
TOTAL DIAS RODAJE:		TFNO. CONTACTO:	
DIRECTOR DE FOTOGRAFIA:		GAFFER:	
EQUIPO TECNICO NECESARIO:		BEST BOY / JEFE ELEC.:	
CARGA		KEY GRIP:	
RODAJE			
DEVOLUCION		NECESIDADES ESPECIALES	
TRANSPORTES NECESARIOS:			
GRUPOS ELECTRÓGENOS:			
CONSUMIBLES:			

PLANTEAMIENTO DE RODAJE:

MATERIAL ILUMINACION:

MATERIAL MAQUINISTA:

OBSERVACIONES:

Iluminación Cinematográfica FM, S.A. - A-78485331Avda. de la Industria 52 -28108 Alcobendas - Madrid. Tel. +34 91 661 06 25 - Fax +34 91 662 56 89 - info@iluminacionfm.comwww.iluminacionfm.com

ANEXO II

PRUEBAS DE USUARIO FINAL

Operación	Caso de uso	Pantalla	Prueba	Resultado	Estado
1	SignIn	**Registro**	Email vacio, password vacio y pulsar botón "Sign In"	Aparece el mensaje "Campo Requerido" en los campos email y password	OK
2			Email sin @ y password vacio y pulsar botón "Sign In"	Aparece el mensaje "Email no válido" en el campo Email y "Campo Requerido" en el campo Password.	OK
3			Email bien formado y password vacio y pulsar botón "Sign In"	Aparece el mensaje "Campo Requerido" en el campo Password.	OK
4			Email bien formado y longitud del password >0 y < 4 y pulsar botón "Sign In"	Aparece el mensaje "Campo Demasiado corto" en el campo Password.	OK
5			Email bien formado y longitud del password > 4 y pulsar botón "Sign In"	Se inicia sesión correctamente y con rol administrador la aplicación navega a la actividad "AdminMainActivity".	OK
6			Email bien formado y longitud del password > 4 y pulsar botón "Sign In"	Se inicia sesión correctamente y con rol empleado la aplicación navega a la actividad "EmployeeMainActivity".	OK
7			Email bien formado y longitud del password > 4 y pulsar botón "Sign In" pero hay pérdida de conexión	Aparece dialogo un mesanje "No hay conexión a Internet"	OK
8			Pulsa la opción "¿Ha olvidado su password?"	Aparece dialogo con un formulario con el campo email a introducir por el usuario.	OK
9	RetrievePass		Email vacio y pulsar botón "Aceptar"	Aparece un mensaje al lado del campo "Campo vacio o demasiado corto"	OK
10			Email sin @ y pulsar botón "Aceptar"	Aparece un mensaje al lado del campo "Email no válido"	OK
11			Email con @ pero sin . Seguido de dos caracteres	Aparece un mensaje al lado del campo "Email no válido"	OK
12			Email bien formado y pulsar botón "Aceptar"	Se envia la petición al servidor.	OK
13			Email bien formado y pulsar botón "Aceptar" pero se pierde la conexión	Aparece dialogo un mesanje "No hay conexión a Internet"	OK
14	GetDocuments	**Principal Administrador**	Se piden Partes de localización tecnica y se produce un problema (Ej. de conexión)	Aparece dialogo un mesanje "No hay conexión a Internet" y un botón para permitir volver a cargar.	OK
15			Se piden los Partes de Localización Técnica y no se produce ningún error.	Se muestran de forma resumida los partes de localización tecnicas ordenados en tiempo de creación y por "Nuevos" y "Vistos".	OK

Operación		Caso de uso	Pantalla	Prueba	Resultado	Estado
	16			Se recorre la lista de partes de localización Tecnica y cuando se llega al último elemento se solicitan nuevos partes	Carga los nuevos partes al final de la lista	OK
	17			Se recorre la lista de partes de localización Tecnica y cuando se llega al último elemento se solicitan nuevos partes pero en ese mismo momento hay una pérdida de conexión.	Aparece dialogo un mensaje "No hay conexión a Internet" y un botón para permitir volver a cargar.	OK
	18			Se pulsa a uno de los elementos de la lista de Partes	Se navega hacia la pantalla donde se muestra los detalles del Parte de Localización Técnica	OK
	19	GetDocumentDetails	**Detalles Parte de Localización Técnica**	En el momento de la petición de los detalles del Parte de Localización Técnica se produce un error de conexión	Aparece dialogo un mesanje "No hay conexión a Internet" y un botón para permitir volver a cargar.	OK
	20			Se pulsa al botón de "Volver a cargar" y se recupera la conexión	Se muestra todos los detalles del Parte de Localización Técnica y además aparece un botón para poder elminiar el parte.	OK
	21			Mostar campos del parte de localización técnica que estan vacios.	Esos campos se mostarán con el string "vacio"	OK
	22	DeleteDocument		Se pulsa al botón de eleminación del Parte de Localización Técnica	Se muestra un mensaje "Se ha elminado el parte de localización Técnica" con un botón "Aceptar"	OK
	23			Se pulsa al botón de eleminación del Parte de Localización Técnica y se produce un error (Ej. Conexión)	Aparece dialogo un mesanje "No hay conexión a Internet".	OK
	24			Tras pulsar el botón "Aceptar" del mensaje de eliminación correctadel Parte de Localización Técnica. Se retorna a la pantalla principal del administrador y se elimina de la lista el documento.	Se elimina el documento de la lista de Partes de Localización Tecnica.	OK
	25			Si se ha consultado un Parte de Localización Técnica hasta la hora no consultado por el usuario y se vuelve a la lista de Partes de Localización	Se reordena la lista de Partes pasando dicho item a la sección de "Vistos".	OK
	26	GetUsers	**Principal Administrador**	Se pide lista de usuarios y se produce un problema (Ej. Conexión)	Aparece dialogo un mesanje "No hay conexión a Internet" y un botón para permitir volver a cargar.	OK

Operación		Caso de uso	Pantalla	Prueba	Resultado	Estado
	26	GetUsers	**Principal Administrador**	Se pide lista de usuarios y se produce un problema (Ej. Conexión)	Aparece dialogo un mesanje "No hay conexión a Internet" y un botón para permitir volver a cargar.	OK
	27			Se pide lista de usuarios y no se produce un problema	Se muestra una lista de usuarios diferenciados por "Administradores" y "Empleados" y ordenados por tiempo de creación.	OK
	28			Se recorre la lista usuarios y cuando se llega al último elemento se solicitan nuevos usuarios.	Carga nuevos usuarios al final de la lista	OK
	29			Se recorre la lista de usuarios y cuando se llega al último elemento se solicitan nuevos usuarios pero en ese mismo momento hay una pérdida de conexión.	Aparece dialogo un mesanje "No hay conexión a Internet" y un botón para permitir volver a cargar.	OK
	30	DeleteUser		Se pulsa el icono de elminación de un usuario y no se produce ningún error.	La lista de usuarios se actualiza eliminado el usuario.	OK
	31			Se pulsa el icono de elminación de un usuario y se produce algún error (Ej. De conexión).	Aparece dialogo un mesanje "No hay conexión a Internet"	OK
	32	ChangePassUser	**Registro**	El primer registro que hace un uusario en el sistema aparecerá en la aplicación un dialogo para cambiar la contraseña.	Aparece un diálogo solicitandole una nueva contraseña y su confirmación.	OK
	33			La nueva contraseña vacia y confirmación de contraseña también	Aparece un mensaje de error "Campos vacios"	OK
	34			La nueva contraseña con < 6 caracteres.	Aparece un mensaje de error "Campo Incorrecto"	OK
	35			La nueva contraseña y confirmación con > 6 caracteres e iguales pero solo minusculas.	Aparece un mensaje de error "La contraseña necesita al menos tener una mayúscula"	OK
	36			La nueva contraseña y confirmación con > 6 caracteres e iguales pero solo minusculas y mayusculas.	Aparece un mensaje de error "La contraseña necesita al menos tener un carácter especial i.e. .!,@,#,etc"	OK
	37			La nueva contraseña y confirmación con > 6 caracteres e iguales pero solo alguna minusculas, alguna mayusculas y algún caracrer especial.	Aparece un mensaje de error "La contraseña necesita al menos tener un número"	OK
	38			Contraseña y confirmación válidas se solicita el cambio de contraseña y no se produce ningún error.	Aparece un mensaje "Cambio de contraseña correctamente " y dependiendo del rol te redirige a "AdminMainActivity" o"EmployeeMainActivity ".	OK

Operación	Caso de uso	Pantalla	Prueba	Resultado	Estado
	39	**Principal Administrador, Principal Empleado, Detalles Parte de Localización Técnica, Actividad Dar de Alta nuevos usuarios y Actividad envio de Partes de Localización Técnica.**	Al pulsar el botón del menú de la aplicación de "Cambiar contraseña"	Aparece un diálogo solicitandole una nueva contraseña y su confirmación.	OK
	41		La nueva contraseña vacia y confirmación de contraseña también	Aparece un mensaje de error "Campos vacios"	OK
	42		La nueva contraseña con < 6 caracteres.	Aparece un mensaje de error "Campo Incorrecto"	OK
	43		La nueva contraseña y confirmación con > 6 caracteres e iguales pero solo minusculas.	Aparece un mensaje de error "La contraseña necesita al menos tener una mayúscula"	OK
	44		La nueva contraseña y confirmación con > 6 caracteres e iguales pero solo minusculas y mayusculas.	Aparece un mensaje de error "La contraseña necesita al menos tener un carácter especial i.e. .!,@,#,etc"	OK
	45		La nueva contraseña y confirmación con > 6 caracteres e iguales pero solo alguna minusculas, alguna mayusculas y algún caracrer especial.	Aparece un mensaje de error "La contraseña necesita al menos tener un número"	OK
	46		Contraseña y confirmación válidas se solicita el cambio de contraseña y no se produce ningún error.	Aparece un mensaje "Cambio de contraseña correctamente "	OK
	47 CreateUser	**Actividad Dar de Alta nuevos usuarios**	Los campos *Email, Nombre, Apellidos, Contraseña, Confirmación de contrase y rol vacios*	El foco se ubica en el campo *Email* hasta que no esta correctamente introducido (con @ . Com, es, etc)	OK
	48		Campo *Email correcto y los campos Nombre, Apellidos, Contraseña, Confirmación de contraseña y rol vacios*	El foco se ubica en el campo *nombre* hasta que no esta correctamente introducido (> 2 caracteres)	OK
	49		Campos *Email y* Nombre *correctos y los campos Apellidos, Contraseña, Confirmación de contraseña y rol vacios*	El foco se ubica en el campo *apellidos* hasta que no esta correctamente introducido (> 2 caracteres)	OK
	50		Campos *Email, Nombre y Apellidos correctos y los campos , Contraseña, Confirmación de contraseña y rol vacios*	El foco se ubica en el campos *Contrase y Confirmación de Contraseña* hasta que no esta correctamente introducido (La contraseña necesita tener al menos 6 caracteres, al menos una mayúscula, una minúscula, un número y un caracter especial i.e. !,@,#, etc.)	OK

Operación	Caso de uso	Pantalla	Prueba	Resultado	Estado
	51		Campos *Email, Nombre, Apellidos, Contraseña y Confirmación de contraseña correctos y el campo rol vacios*	El foco se ubica en el campo *rol* hasta que no esta correctamente introducido (seleccionado una de las dos opciones (Empleado o Administrador))	OK
	52		Todos los campos correctos y se pulsa al botón crear	Aparece un mensaje de confirmación con los botones "Aceptar" y "Cancelar".	OK
	53		Se pulsa al botón "Cancelar" del mensaje de confirmación para crear nuevo usuario.	Desaparece el mensaje y se muestra los campos del nuevo usuario.	OK
	54		Se pulsa al botón "Aceptar" del mensaje de confirmación para crear nuevo usuario y se envia petición al servidor sin producirse ningún fallo.	Aparece un mensaje de éxito "Usuario con nombre: [nombre introducido] cerado correctamente"	OK
	55		Se pulsa al botón "Aceptar" del mensaje de confirmación para crear nuevo usuario y se envia petición al servidor y se produce algún fallo.	Aparece un mensaje de error "No hay conexión a Internet" , "Email ya registrado" o "Campos incorrectos"	OK
	56 CreateDocument	**Actividad envio de Partes de Localización Técnica.**	Los campos obligatorios *Prdouctora, producción, localización, fecha de rodaje, total dias rodaje, equipo técnico necesario, carga, rodaje, devolución, rpble de produccion, mail de contacto, gaffer, best boy, key grip vacios.*	Se irán pidiendo (foco al campo) que se introduzcan si el usuario da al botón de envío.	OK
	57		Campos obligatorios introducidos correctamente y se pulsa al botón de enviar	Se muestra mensaje de confirmación con los botones "Cancelar" y "Aceptar".	OK
	58		Si se pulsa al botón "Aceptar" en el mensage de confirmación de creacion de Parte de Localización Técnica y no se produce ningun error	Aparece mensaje de éxito "Documento creado correctamente" con el botón "Aceptar"	OK
	59		Si se pulsa al botón "Aceptar" en el mensage de confirmación de creacion de Parte de Localización Técnica y se produce algún error	Aparece un mensaje de error "No hay conexión a Internet" o "Error al crear documento"	OK
	60 LogoutUser	**Principal Administrador, Principal Empleado, Detalles Parte de Localización Técnica, Actividad Dar de Alta nuevos usuarios y Actividad envio de Partes de Localización Técnica.**	Al pulsar el botón del menú de la aplicación de "Salir"	Aparece un diálogo pidiendo una doble confirmación	OK
	61		Se pulsa "Aceptar" en la doble confirmación para cerra la sesión	Se redirige al usuario a la pantalla de registro y se produce la finalización de sesión	OK
	62		Se produce un error en el cierre de sesión en la pantalla de registro.	Se muestra el error "Error al cerrar sesión" o "No hay conexión a Internet" junto a un boton "Aceptar"	OK
	63		Se pulsa el botón "Aceptar" del mensaje de error de finalización de sesión	se redirige al usuario dependiendo de su rol a su pantalla principal	OK